

RADC-TR-88-324, Vol III(of nine)
Interim Report
March 1989



AD-A208 379

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1987 Distributed Artificial Intelligence for Communications Network Management

Syracuse University

Robert A. Meyer and Susan E. Conry

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700**

DTIC
ELECTE
MAY 24 1989
S H D

89 5 24 078

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

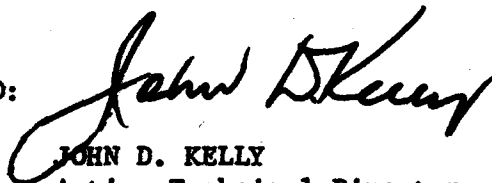
RADC-TR-88-324, Vol III (of nine) has been reviewed is approved for publication.

APPROVED:



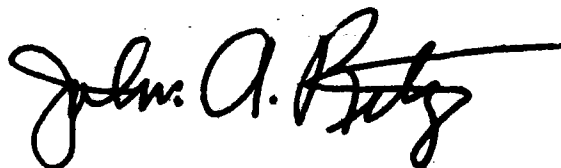
ARLAN MORSE, Capt, USAF
Project Engineer

APPROVED:



JOHN D. KELLY
Acting Technical Director
Directorate of Communications

FOR THE COMMANDER:



JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (DCLD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-324, Vol III (of nine)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (DCLD)		
6a. NAME OF PERFORMING ORGANIZATION Northeast Artificial Intelligence Consortium (NAIC)		6b. OFFICE SYMBOL (if applicable)		7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	
6c. ADDRESS (City, State, and ZIP Code) 409 Link Hall Syracuse University Syracuse NY 13244-1240		8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (if applicable) COES	
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008			
10. SOURCE OF FUNDING NUMBERS					
PROGRAM ELEMENT NO.		PROJECT NO.		TASK NO.	
33126F		2155		02	
10. SOURCE OF FUNDING NUMBERS		WORK UNIT ACCESSION NO.			
33126F		10			
11. TITLE (Include Security Classification) NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1987 Distributed Artificial Intelligence for Communications Network Management					
12. PERSONAL AUTHOR(S) Robert A. Meyer, Susan E. Conry					
13a. TYPE OF REPORT Interim		13b. TIME COVERED FROM Dec 86 to Dec 87		14. DATE OF REPORT (Year, Month, Day) March 1989	
15. PAGE COUNT 114					
16. SUPPLEMENTARY NOTATION This effort was performed as a subcontract by Clarkson University to Syracuse University, Office of Sponsored Programs.					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Artificial Intelligence, Distributed Artificial Intelligence,		
12	05		Simulation, Distributed Planning, Graphical User Interface,		
12	07		Knowledge-based Reasoning, Communications Network		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the third year of the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, dis- tributed AI for communications system control, automatic photo interpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware archi- tectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system. The specific topic for this volume is the use of knowledge-based systems for communications network management and control via an architecture for a diversely distributed multi-agency system.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL ARLAN MORSE, Capt, USAF			22b. TELEPHONE (Include Area Code) (315) 330-7751		22c. OFFICE SYMBOL RADC/DCLD

NAIC

Northeast Artificial Intelligence Consortium
1987 Annual Report

VOLUME 3

DISTRIBUTED ARTIFICIAL INTELLIGENCE FOR COMMUNICATIONS NETWORK MANAGEMENT

Robert A. Meyer
Susan E. Conry

Electrical and Computer Engineering Department
Clarkson University
Potsdam, NY 13676

3 Table of Contents

3.1	Executive Summary	3-2
3.2	Introduction	3-4
3.3	Environment for Development of Distributed Systems	3-6
	3.3.1 System Structure	3-6
	3.3.2 Concurrency Control	3-7
	3.3.3 Interface Facilities and Performance Issues	3-8
3.4	Distributed Planning	3-10
	3.4.1 Plan Characteristics	3-10
	3.4.2 Plan Generation	3-11
	3.4.3 Reasoning About Constraints and Conflicts	3-13
	3.4.4 Multistage Negotiation	3-19
3.5	Distributed Knowledge Base Management	3-22
	3.5.1 Introduction	3-22
	3.5.2 Representational Issues in Domain Knowledge	3-22
	3.5.3 Distribution of Global Network Knowledge	3-24
	3.5.4 Management of a Distributed Knowledge Base	3-26
	3.5.5 Future Work	3-29
	Bibliography	3-30
Appendix 3-A SIMULACT Users Manual (Draft)		
Appendix 3-B A Distributed Development Environment for Distributed Expert Systems		
Appendix 3-C SIMULACT, A Generic Tool for Simulating Distributed Systems		
Appendix 3-D The Role of Knowledge-based Systems in Communications System Control		
Appendix 3-E Machine Intelligence for DoD Communications System Control		

NAIC Annual Report

Volume 3

3.1 Executive Summary

The work described in this volume has been performed at Clarkson University during 1987, the third year of the NAIC research contract with the Rome Air Development Center. The AI research at Clarkson has continued to concentrate on the study of distributed problem solving systems, using communications network management and control as the problem domain. This report gives a brief introduction to the problems of interest and a short review of work completed in the previous years. The major portion of this report documents the principal research accomplishments of this year.

Activity at Clarkson has focused in three areas: distributed planning, distributed knowledge base management, and an environment for the development of distributed systems. Significant progress has been achieved in distributed planning, and our distributed development environment has matured. Work on a distributed knowledge base manager (KBM) has been driven, to some extent, by the requirements for knowledge base access generated from the planner.

Planning work has centered around further investigation of multistage negotiation as a cooperation paradigm for distributed constraint satisfaction problems. In our domain these problems arise from the need to restore service for multiple disrupted communication circuits in a network with decentralized control and limited localized knowledge about global network resource availability. Two aspects of negotiation have been explored. The first is the phase involving plan generation. This has been especially interesting in that new insight has been gained concerning the degree of nonlocal knowledge necessary in formulating feasible local choices.

Work in distributed planning has also led to formulation of an algebra of conflicts to be used in reasoning during negotiation. The planning problems we find in the communications systems domain involve distributed resource allocation problems in which hard local constraints must be enforced in a global plan. This year, we have devised mechanisms which enable an agent to reason about its own local constraints and exchange knowledge concerning the impact of nonlocal actions, incorporating new knowledge in a consistent manner. These mechanisms appear to be particularly interesting because they seem to be applicable to a wide class of distributed constraint satisfaction problems.

During 1987, SIMULACT, our distributed development environment has matured. It has been used as a testbed environment in investigations of planning issues and in managing distributed knowledge base functions. Over the course of the year, SIMULACT has been

converted to Common Lisp, enhanced user interface facilities have been incorporated, and the system has been implemented on a network of Lisp machines. Experiments have been performed to determine the extent to which process management overhead impacts apparent distributed system performance in single processor and multiple processor configurations. In addition, SIMULACT has been demonstrated running distributed applications at the AAAI Annual Meeting and at the NAIC meetings.

Work on the Knowledge Base Manager has been driven largely by needs perceived in distributed planning. Much of the progress has been in identifying the forms of interface and query processing required by the planner. In addition, we have investigated the feasibility of a natural language user interface, primarily as a debugging tool. While designing the KBM, we uncovered certain limitations arising from the knowledge representation scheme initially chosen for communications circuits. A moderate set of revisions to the design of the knowledge base has been completed as a result of changing the representation of circuit knowledge. These changes were also incorporated into GUS, a Graphical User interface for Structural knowledge; which we developed in previous years as a tool for knowledge acquisition in the communications network domain.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or special
A-1	

3.2 Introduction

The goal of this research is to answer fundamental questions about the design of distributed, cooperative, knowledge-based problem solving systems. Our research is conducted with the context of a particular problem domain in mind; this domain is communication network management and control. During the previous two years we have performed an in-depth analysis of this problem domain, relying primarily on a model of the Defense Communications System (DCS) as it is envisioned for the late 1990's. The result of this effort was the design of a high-level architecture for a *diversely distributed, multi-agent* knowledge-based system. We view such a system as consisting of a distributed collection of semi-autonomous, cooperating specialists which provide assistance to human operators.

During this past year we prepared and presented two papers at technical meetings which document our problem domain analysis. In the first [1] (also in Appendix 3-D of this report), we describe the functional requirements as determined from our analysis. These requirements form the basis for the system architecture. From an AI perspective, our proposed architecture is unique in its need for distribution in both the *spatial* or *geographic* domain and in the *functional* domain. As the system is being designed and implemented we are discovering new techniques to handle multi-agent interaction under varying assumptions concerning similar goals and problem solving strategies among the agents.

A second paper [2] (also in Appendix 3-E) relates our work, which is intended to be basic engineering research or very early development, to other research efforts oriented to specific prototype development of expert systems for communications network control. We believe one strength of our work is that it addresses fundamental distributed AI research issues in the context of current, real-world problems. In [2] we discuss how these projects together form an evolutionary design and development path for the transition of AI technology from research labs to field usage.

One of our first objectives was to create an environment for the development of distributed AI systems. We are interested in distributed systems of the "coarse-grained" variety, having a relatively small number of processors. However, even for numbers in the range of 5 to 25 processors, it is very expensive to provide a dedicated research facility with a network of Lisp machines. Instead, we have developed SIMULACT, a simulation facility for multiple actors designed to operate on a network of one or several Lisp machines. SIMULACT has several important characteristics which have been documented in detail in the SIMULACT User's Manual (in Appendix 3-A). A general introduction and summary of SIMULACT's features are given in section 3.3 of this report. The design of SIMULACT and its performance characteristics have been documented in two technical papers presented during this year [3,4] (also in Appendices 3-B and 3-C).

Early in the development of our distributed testbed we recognized the need for a tool to assist in the acquisition of large amounts of rather tedious knowledge about the

structural aspects of a communications network. This structural knowledge is very important because much of the reasoning needed to assess network state, to diagnose faults, and to generate restoral plans is based on the underlying network structure. We also found it necessary to acquire this knowledge on a global network-wide scale, even though each local network controller would have a limited, local view of the network. During the two previous years a tool was developed for this purpose. This program, called GUS, uses a graphical interface to acquire knowledge from the user about network structure. The initial version of GUS was completed and documented in a thesis [5] in January of this year. During this year we made revisions, additions and bug fixes to GUS as well as the changes necessary to maintain compatibility with the current release of the Symbolics Lisp System.

The major concentration of our effort this year has been on the design and implementation of a distributed planner and a distributed knowledge base manager (KBM). The planner is at a more advanced stage because it continues the research started last year which resulted in the development of multistage negotiation. During this year we implemented the first version of the plan generation phase and began to refine the design of the multistage negotiation protocol. A key idea of multistage negotiation is that an agent of a distributed planner must be able to discover conflicts among subgoals which arise from the interaction of local and nonlocal constraints. We have developed a formal system for reasoning about conflicts and propagating the impact of local decisions among the agents involved. This is done *without* the exchange of complete local state. Section 3.4 gives a more detailed presentation and includes an extensive example illustrating this process.

The design of a distributed KBM has followed the implementation of the planner. At each local network controller a knowledge base must be maintained with the locally known facts, beliefs and conclusions derived about the operational state of the network. One of the functions performed by the KBM is to respond to local queries from other agents at the same site, such as the planner. Thus, as the planner has progressed from design to implementation, a corresponding part of the KBM has also been developed. This interdependence between KBM and the planner has contributed to our gaining a better understanding of representational issues associated with some forms of domain knowledge. For example, we found that our initial understanding of circuits was inadequate for use with the problem solving paradigm of the planner. This is explained in more detail in section 3.5. Much work remains to be done in the design of a fully functional distributed KBM. Our research directions in this area for the next year are also described in section 3.5.

3.3 Environment for Development of Distributed Systems

In this section, we describe our continuing work with SIMULACT, a development environment for distributed problem solving systems. During 1987, most of our attention on SIMULACT has been focused in two areas: development of better user interface facilities and distribution of the system so that multiple host networks can be utilized. In this section, we briefly outline various features of SIMULACT and indicate the nature of the performance results. More detail (including a user's manual) is found in Appendices 3-A, 3-B, and 3-C.

The underlying model of problem solving which is employed in SIMULACT regards the problem solving system as a collection of semi-autonomous agents which cooperate in problem solving through an exchange of messages. The system is modular: each agent is essentially an independent module which can easily be "plugged in" to the system. An agent's interaction with other agents in the system is totally flexible, and is user specified. Neither the form nor the content of inter-agent messages is specified by SIMULACT itself. In addition, the user can suspend execution at any time, examine the state of any agent, modify the state, the knowledge base, or even the code of an agent, and resume execution. A trace facility makes post-mortem examination of activity feasible, and a gauge facility allows the user to instrument the system in a very flexible manner.

SIMULACT is a distributed system that allows n agents to be modelled on k machines, where $n > k$. Each agent runs asynchronously and coordinates its activity with that of other agents through the exchange of messages. The activities performed by each agent are assumed to be complex, so that the parallelism is coarse grained. SIMULACT allows the programmer to write code in Lisp as though there were as many Lisp machines in the network as there are agents in the distributed system being developed.

3.3.1 System Structure

SIMULACT is comprised of four component types: Actors, Ghosts, Directors, and an Executive Director. Actors are used to model agents in the distributed environment. Each Actor type is individually defined, and used as a template to create multiple instances of that Actor type. An Actor is a self contained process which runs in its own non-shared local environment. Although Actors run asynchronously, the elapsed CPU time for each actor never varies by more than one "time frame".

Ghosts are used in SIMULACT to generate and inject information into the model that would naturally occur in a "real" distributed system. They do not represent any physical component of the model. For example, external inputs (alarms, sensors, etc.) affecting the state of the system can be introduced via Ghosts, as well as inputs that reflect the "side effects" of the system's activities. Ghosts can also be used to inject noise or erroneous information into the system so that issues concerning robustness can be easily investigated. The performance of an knowledge based system can be monitored in subsequent runs through the simple modification of these Ghosts.

Due to the similarities between Actors and Ghosts, we refer to them as Cast members. Each Cast member has a unique "stagename" and a "mailbox" used by the communication facility in routing messages among members. Each also has a "script function" which defines its high level activity.

The control structure residing at each host processor in SIMULACT's distributed environment is known as the Director. The Director is responsible for controlling the activities of the Cast members at that site, and for routing messages to and from these members. These activities are assigned to the Grip and Messenger respectively. The responsibilities of the Grip range from setting up and initializing each Cast member's local environment to managing and executing the Actor and Ghost queues. The Messenger only deals with the delivery and routing of messages. When a message is sent, it is placed directly into the Messenger's "message-center". During each time frame, the Grip invokes the Messenger to distribute the messages. Whenever the destination stagename is known to the Messenger, the message is placed in the appropriate Cast member's mailbox. Otherwise, it is passed to the Executive Director's Messenger and routed to the appropriate Host.

There is one Executive Director in SIMULACT which coordinates all Cast member activities over an entire network. The Executive Director provides the link between Directors necessary for inter-machine communications, directs each Grip so that synchronization throughout the network is maintained, and handles the interface between the user and SIMULACT.

3.3.2 Concurrency Control

Concurrent execution of n Actors on k machines ($n > k$) is emulated through the imposition of a "time frame" structure in execution. A time frame cycle is divided into three phases: invocation of the Ghosts, the distribution of mail by the Messengers, and invocation of the Actors.

At the start of the first time frame, the Executive Director notifies all Directors to begin executing Ghosts. (This models the occurrence of events in the world external to the distributed system.) At the conclusion of the Ghost frame, each Director automatically invokes its Messenger. The Messenger distributes all messages which were generated during the current Ghost frame, as well as all those resulting from the previous Actor frame. Mail destined for Cast members residing on the same host processor is placed in the appropriate mailboxes and all non-local mail is forwarded to the Executive Director's Messenger. In order to reduce network overhead, this transfer is done in the form of a single message. This communication always occurs, even if there are no messages to distribute, as a synchronizing mechanism for the time frame so that Actors cannot "run away". After sending this message, each Director enters a wait state until an Actor frame directive is received from the Executive Director. The Executive Director's Messenger is invoked immediately following the receipt of the last Director's Messenger communication.

Upon receiving an Actor frame command from the Executive Director, each Director's Messenger is invoked to distribute any inter-machine messages that may have been received. Next, each Actor is allowed to run for one time slice (time frame). At this point the Executive Director immediately enters its next time frame cycle, sends the Ghost frame command, and waits for all the Director Messengers to send their next synchronizing signal.

3.3.3 Interface Facilities and Performance Issues

There are five user interface facilities available in SIMULACT. These facilities provide mechanisms for inter-agent communication (Mail), code sharing (Support Packages), interactive monitoring and debugging (Peek and Poke), post mortem trace analysis (Diary), and runtime monitoring (Gauge). They were designed to make SIMULACT more attractive as a development environment for knowledge based systems. Each of these facilities is explained in more detail in Appendix 3-A and Appendix 3-B

The overhead incurred in managing the emulation of a distributed environment is one important measure of system performance. We have conducted a group of experiments to assess the degree of overhead incurred by SIMULACT. These experiments were designed to obtain results that would assess SIMULACT's behavior as the number of messages per time frame increases. In each of the experiments, the number of messages per time frame, m , was varied over the range 0 to $10n$, where n is the number of Actors in the system. Each Actor process worked continually, consuming its total time slice allowed per time frame. Thus when $n = 0$, we measured SIMULACT's best case performance. It should be pointed out that in the distributed case where $n > 0$, the number of messages per time frame in our experiments represented entirely inter-machine communications, emulating a worst case scenario.

The measurement used to represent SIMULACT's performance was a "time frame ratio" gauge. This ratio is defined as:

$$\frac{(\text{elapsed wall time})}{(\text{sum of all Actor elapsed time})}$$

This ratio times the number of Actors in the system provides an estimate of how much time is required by SIMULACT to execute one time frame. For the ideal situation involving no overhead, this ratio would be 1.0 and 0.5 for the one and two machine cases respectively.

For the single machine case with no message passing, SIMULACT's overhead approaches 4.5%. Similarly, the distributed case approaches 10% overhead. Both sets of data indicate that as the number of messages per time frame increases, so does the overhead. In fact, between 100 and 200 messages per time frame handled by the system seems to be a saturation point for the Messenger. Currently the Messenger uses an a-list to associate stagenames with Cast members. We should see improvement when this is implemented as a hash table lookup. Also note that the distributed case after saturation degrades at a

much faster rate. One explanation for this can be found in the observation that all inter-machine messages are handled three times by different Messengers and must be sent over the Lisp machine network. Messages among agents residing at the same host processor are handled once by the Messenger and sent directly to the appropriate mailbox.

3.4 Distributed Planning

In this section, our research in distributed planning is described. We view distributed planning as a task which is carried out by a group of semi-autonomous agents, each of which has a limited view of the global system state and control over only a subset of the resources required to determine and execute an acceptable plan. Our model of distributed planning is therefore one involving two phases: a plan generation phase and a negotiation phase.

Because our model of distributed planning is one which can be applied to domains in which planning involves distributed allocation of scarce resources, we first discuss the characteristics which plans in this type of domain have. We then provide more detail concerning the plan generation phase and our solution to some of the problems which arise in plan generation in section 3.4.2. For negotiation to be effective as a mechanism for distributed decision making, agents must be able to reason about the nonlocal impact of locally attractive decisions. We have developed a formalism for use in reasoning about local constraints and the impact of nonlocal decisions on the set of local constraints. These are discussed in section 3.4.3. Finally, in section 3.4.4, we describe the process of multistage negotiation as a protocol which uses this formalism as the basis for reasoning in determining an acceptable plan.

As has been mentioned, plan generation is a stage in which alternative sub-plans are determined for satisfaction of each global goal that has been instantiated. It involves a distributed search across problem solving agents for coordinated resource allocations that could be used to satisfy global goals. During plan generation, each global goal is decomposed into local subgoals each of which represents requests to partially satisfy that goal using local resources. Each agent has knowledge concerning parts of global plans that use resources local to that agent. Negotiation then determines a set of plans for execution which forms at least a satisficing solution for the system goal of satisfying a maximal number of global goals.

3.4.1 Plan Characteristics

In many domains, planning can be viewed as a distributed resource allocation problem in which resources are objects that are available for use in completing some task. The resources available have two significant characteristics: resources are indivisible (not consisting of component resources), and the supply of resources is limited.

Our model of planning differs from many others in that both control over resources and knowledge about these resources are distributed among problem solving agents. Some of the resources are under the direct control of a single agent, while control over others is shared by two agents. Resources controlled by a single agent are local to that agent and cannot be allocated by any other agent. Indeed, any given agent only has knowledge concerning those resources that are local and those it shares with other agents. Although shared resources are also locally known, they must be regarded

somewhat differently in reasoning because allocation of shared resources must be coordinated by those agents which share control. Each agent must therefore know which of its resources are shared and which agents are involved in the shared control of a particular resource.

In this kind of environment, a plan is a sequence of local resource allocations which satisfy some number of global goals. In general, global goals arise concurrently in multiple agents. An acceptable plan is thus a set of resource allocations which satisfies as many of the global goals as possible, subject to local resource constraints. As has been mentioned, planning proceeds in two phases: plan generation and negotiation. Plan generation is a stage in which alternative sub-plans are determined for satisfaction of each global goal that has been instantiated. This phase is analogous to finding all sequences of operator/argument pairs that could be used to accomplish a task in classical planning systems. Plan generation involves a distributed search across problem solving agents for coordinated resource allocations that could be used to satisfy global goals.

Plan generation determines a set of plans each of which satisfies some individual global goal. Each of these plans is feasible, taken in isolation, since plan generation does not consider any goal interaction problems. Negotiation is necessary to select a set of plans which satisfies as many global goals as possible while not violating any of the constraints which are applicable.

In our application domain, service restoral is treated as a distributed planning problem. Each global goal corresponds to the restoral of a single circuit that has been disrupted. A resource is a channel on a trunk, and a plan is a sequence of trunk connections to restore a path between two endpoints. Each agent has control over all resources in an entire subregion and shares those resources that cross subregion boundaries. The reader should refer to section 3.5 for an explanation of these domain specific terms.

3.4.2 Plan Generation

During plan generation, each global goal is decomposed into local subgoals which represent requests to partially satisfy the global goal using local resources. Thus, an agent knows about parts of "global" plans that use resources local to that agent. These partial plans are called plan fragments. Each subgoal in an agent has associated with it all the plan fragments the agent could use to partially satisfy the global goal corresponding to that subgoal. As a result of this decomposition, plans exist only in distributed form, as plan fragments distributed among the agents. It is the shared resources that provide the connections between plan fragments in various agents to form global plans.

As we have previously noted, plan generation involves a distributed search among agents. It is important to realize that plan generation has a number of characteristics which distinguish it from standard circuit routing problems, so it is not possible to

use standard algorithms associated with these problems. Circuit routing algorithms generally attempt to restore one circuit at a time by finding the "best" path as measured by some cost function. In contrast, we attempt to restore multiple circuits at a time through negotiation over available alternatives. Since negotiation requires that multiple paths be found for each circuit, the notion of using routing tables which establish preferred next neighbors is not applicable. Algorithms that depend upon global routing information at each node are also impractical in our domain due to the cost of maintaining consistency among nodes. When viewing graph representations of the standard problems, it is usually assumed that a path exists between any two edges into a vertex. In our domain, however, the vertices correspond to subregions. In performing service restoral we make certain assumptions about network connectivity. Within a site, we assume that it is possible to complete a path between any two links entering the site. However, we do not assume the same connectivity between links entering a subregion. It may be impossible to connect two links entering a subregion using only resources within that subregion. As a consequence, it may not be possible to connect two sites in a subregion using only local resources.

Plan generation begins when an agent is notified that a global goal has been instantiated. The agent creates a subgoal corresponding to this global goal and derives all the alternative ways it can use its resources in partial satisfaction of this subgoal. Each of these alternatives becomes a plan fragment. If any of these plan fragments involve shared resources, the appropriate agent is sent a request to continue building the plan using its local resources. This process is repeated until all requests to build the plan have been answered.

Each request issued while building a plan must carry enough information to permit construction of a complete plan in satisfaction of the global goal. Agents which subsequently handle requests for satisfaction of a particular goal must be able to determine which of their local resources could be used to extend the plan. They must also be able to ascertain which plan fragments are associated with satisfaction of the same global goal. For these reasons, requests for extension of partially constructed plans must contain an identifier for the associated global goal, a description of that goal, and the name of shared resources involved in the extension of the plan (together with any constraints on these shared resources).

It should be noted that it is possible for an agent (agent A) to have partially constructed a plan and pass responsibility to another agent for extension of that plan only to have some third agent request that agent A extend the same plan further. For this reason, agents must be able to detect when they are being asked to extend a plan which they have partially constructed and determine which plan fragment is associated with the requested extension. This is necessary because the associated plan fragment will be augmented as the plan is extended. The strategy of augmenting existing plan fragments in this way provides a mechanism whereby the inter-agent negotiation can be made more effective, reducing the extent to which redundant work is performed. The strategy is only feasible if it is possible to match existing plan fragments in an agent with requests which would extend them. Our solution to the problem involves a naming scheme for plans in which each agent appends a unique name to each plan fragment which

can be used to extend a plan. This name is carried with the plan as it is developed.

Since it is possible for more than one agent to be notified of the instantiation of the same global goal, the search strategy can be improved. When an agent receives a request to continue building a plan, it checks the plan fragments it has already constructed for the same global goal. If the agent can use a partial plan it has already built to complete the incoming plan, it matches these partial plans in satisfaction of the request.

There are two possible replies for a request to continue building a plan. If an agent has no way to continue the plan or if none of the corresponding plan fragments have satisfactorily completed the plan, then the requesting agent is notified to delete the corresponding plan fragments as viable alternatives. On the other hand, if an agent is able to complete the request, either solely or with the aid of other agents, then the requesting agent is notified that the corresponding plan fragments are parts of a feasible plan. It is important to note that when two agents share a resource, each may view that resource differently. When one agent must constrain how a shared resource is used in the plan, then the requesting agent is also notified of the constraint.

In the context of our domain, the purpose of plan generation is to determine all loop-free paths that could be used to restore a circuit. When a circuit fails, plan generation is initiated in the two subregions where the circuit terminates. In issuing requests to build a plan, the agents pass a goal description which includes the name, priority, source, and destination of the circuit to be restored. Agents derive plan fragments by requesting information about the physical world from the local knowledge base manager.

3.4.3 Reasoning About Constraints and Conflicts

Subgoal interaction problems are of critical importance in conventional planning systems. Reasoning about these problems is essential in determining a feasible plan of action in most cases. In distributed planning systems, detecting and handling subgoal interactions is just as important as in more conventional systems and even more difficult. Multistage negotiation has been devised as a mechanism whereby agents in a distributed problem solving system can exchange knowledge about nonlocal state and reason about the impact of nonlocal decisions on those made locally. In order to do this, the nature of the subgoal interactions and the character of the reasoning required to arrive at reasonable decisions must be formalized.

During 1987, we have been particularly concerned with determining the nature of goal interactions and devising a way of reasoning about these interactions in a distributed environment. In our application domain, the goal interaction problems manifest themselves in the form of conflicts over resources which have their roots in the constraints on resource utilization which are present in the domain.

In this subsection, we describe a formalism which has been developed for propagating information about nonlocal impact of decisions made locally. The discussion in this subsection is organized around an example. This example is simple, but intended to illustrate the reasons why subgoal interactions occur, the nature of these interactions, and the way in which knowledge concerning nonlocal impact of local decisions can be propagated.

We consider as an example a scenario in which there are four agents (agents A, B, C, and D) in a distributed planning system. In this example scenario, each of these agents has knowledge concerning certain resources. This local knowledge is indicated in Table 3.1. If entry (agent i , resource r) in Table 3.1 is k , then agent i has k copies of resource r to utilize in problem solving. The shared resources (such as $r2$ and $r5$) are evident, as they are known to more than one agent.

	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
Agent A	3	2									2
Agent B									1	1	2
Agent C		2	3	2	2					1	
Agent D					2	2	1	3	1		

System Resource Availability

Table 3.1

This scenario assumes that the system is attempting to simultaneously satisfy four global goals: $g1$, $g2$, $g3$, and $g4$. During plan generation, global plans have been determined for each of these goals. These plans and the resource requirements associated with each are shown in Table 3.2. It should be noted that Table 3.2 shows the global plans from a global perspective. No single agent in a distributed problem solving system has complete knowledge concerning any of these plans. Indeed, as is shown in Table 3.3, no single agent is even aware of the total number of alternative plans that have been generated.

From these two tables, it is evident that global plans are composed of collections of local plan fragments. For instance, global plan $g1p1$ is composed of plan fragments A-a, B-a, C-a, and D-a while global plan $g4p2$ consists of A-h and C-j. Examination of Table 3.2 and Table 3.3 also reveals the type of constraints that are relevant in this distributed planning problem. A choice on the part of agent A to satisfy $g2$ through execution of plan fragment A-d constrains the set of feasible and consistent alternatives known to agent C. Given agent A's choice, agent C must utilize plan fragment C-d if it is to contribute in satisfying $g2$. It is this kind of nonlocal impact of local decisions that must be assessed by an agent in determining its actions.

To enable an agent to efficiently exchange knowledge concerning the nonlocal impact of local decisions, we first determine a conflict set for each plan fragment. The conflict set for plan fragment x indicates the minimum impact (locally) associated with

a choice to execute plan fragment x . If one were to consider a maximal set M of mutually feasible plan fragments (including plan fragment x) known to an agent, the

	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
g1p1	1	1			1	1			1		1
g1p2	1			1	1		1	1			1
g1p3	1	1		1							
g1p4						1		1			
g2p1	1	1	1	1							
g2p2	1		1	1					1		1
g2p3			1		1	1		1			
g3p1	1	1	1		1		1				
g3p2						1	1			1	
g4p1	1	1			1		1			1	1
g4p2		1		1						1	1
g4p3				1	1	1				1	

Global Plans Generated

Table 3.2

complement of M is an element of the conflict set for x . It should be noted that the complement is taken with respect to the set of all of the agent's plan fragments for goals other than the one associated with x . The conflict set for plan fragment x can in fact be defined as the set of sets constructed in this manner by considering the complements of all maximal mutually feasible sets of local plan fragments determined in this way. Though this view of the conflict set is intuitively appealing, it is often more computationally attractive to treat the conflict set of x in its dual form: as the collection of minimal mutually infeasible sets of plan fragments, given that plan fragment x is to be executed.

Three significant observations can be made concerning the conflict set of a plan fragment. First, the complement of each element of the conflict set is indeed a maximal feasible set. Secondly, the agent will be compelled to forego execution of the plan fragments in some element of the conflict set if it chooses to execute plan fragment x . The "badness" of a decision can be related to the size of elements in the conflict set. Finally, representation of impact in the form of a conflict set seems to provide a substantially more compact form of representation that can be more efficiently used in reasoning than many others.

Agent A		r1	r2	r11	Agent B		r9	r10	r11
g1	A-a	1	1	1	g1	B-a	1		1
	A-b	1		1		B-b			1
	A-c	1	1		g2	B-c	1		1
g2	A-d	1	1			B-d			1
	A-e	1		1	g4	B-e		1	1
g3	A-f	1	1						
	A-g	1	1	1					
g4	A-h		1	1					

Agent C		r2	r3	r4	r5	r10
g1	C-a	1			1	
	C-b			1	1	
	C-c	1		1		
g2	C-d	1	1	1		
	C-e		1	1		
	C-f		1		1	
g3	C-g	1	1		1	
	C-h					1
g4	C-i	1			1	1
	C-j	1		1		1
	C-k			1	1	1

Agent D		r5	r6	r7	r8	9
g1	D-a	1	1			1
	D-b	1		1	1	
	D-c		1		1	
g2	D-d					1
	D-e	1	1		1	
g3	D-f	1		1		
	D-g		1			
g4	D-h	1		1		
	D-i	1	1			

Local Knowledge About Plan Fragments

Table 3.3

To illustrate the concept of the conflict set, suppose that agent A chooses to satisfy g_1 through executing plan fragment A-a. Because no agent should act to satisfy a given goal in more than one way, we disregard plan fragments A-b and A-c in constructing the conflict set for A-a. Using the (revised) constraints on resource availability we construct the conflict set. This set is:

$\{(A-d, A-f, A-h), (A-d, A-g, A-h), (A-d, A-e, A-f, A-g), (A-f, A-g, A-h)\}$

From this example, it is clear that each element of the conflict set may indeed contain more than one plan fragment for a given goal. The reason for this phenomenon is that the complement of an element of the conflict set forms a maximal feasible set (locally). Consider, for instance, the set $\{A-d, A-g, A-h\}$. The complement of this element of the conflict set is $\{A-a, A-e, A-f\}$, which is a maximal feasible set. Agent A must, if it chooses to execute A-a, forego execution of all plan fragments present in one of the four elements of the conflict set.

The conflict set is concerned with relationships among plan fragments. Each agent must also be able to assess the impact of a choice on its ability to contribute to satisfaction of the global goals about which it has knowledge. The exclusion set associated with plan fragment x is a collection of sets of goals. This set has the property that if the agent elects to satisfy the goal associated with plan fragment x through execution of x then some element of the exclusion set is a set of goals that cannot be satisfied through action on the part of this agent. Thus a choice to execute plan fragment x excludes some element of the exclusion set as far as this agent is concerned. Given the conflict set for a plan fragment and knowledge concerning the goals associated with each plan fragment which is local to an agent, it is not difficult to compute the exclusion set for that plan fragment.

The example which was discussed above can be used to illustrate this concept as well. Examination of the conflict set for A-a reveals that each element of the conflict set contains either all locally known plan fragments for g_3 or all local plan fragments for g_4 . Thus we may conclude that execution of A-a by agent A excludes either g_3 or g_4 . The exclusion set of A-a is therefore $\{(g_3), (g_4)\}$. It should be noted that the exclusion set only deals with the impact of choosing to execute a single plan fragment.

The exclusion set exposes relationships between plan fragments and goals. It is often desirable to detect and reason about mutually infeasible goals. The relationship of infeasibility is a very strong one. Goal g_1 is (locally) infeasible with goal g_2 if all (local) plan fragments for g_1 exclude g_2 and conversely. Once exclusion sets have been determined, infeasibility is not difficult to detect.

The three types of relationships we have presented are all rooted in local constraints. Conflict, exclusion, and infeasibility are essentially concepts which would not be relevant were it not for the constraints on joint execution of plan fragments that exist locally. Although the concept of conflict does not appear to propagate in a meaningful manner, those involving exclusion and infeasibility do. The

key element in this propagation lies in the observation (which we have made before) that a choice on the part of one agent to satisfy a goal through execution of a specific plan fragment constrains the set of remaining choices that are available to other agents.

We have developed a notion of nonlocal or induced exclusion which captures the essence of the impact which local decisions have nonlocally. In a distributed environment, one agent does not have knowledge concerning another agent's internal state. It specifically does not have any knowledge about resources that are not local to it. The agent must request information about the impact its choice has on other agents.

The induced exclusion set is incrementally built during negotiation. When one agent (agent A) requests information about the impact of executing plan fragment x on another agent (agent B), agent B attempts to summarize all the knowledge it has about that impact. This knowledge is initially found in the exclusion sets of each of its plan fragments which match plan fragment x . Initially, then, the induced exclusion set which is built at agent A is empty. As nonlocal knowledge becomes available, this set is augmented. Suppose that agent B knows of two plan fragments (call them plan fragments y and z) which match plan fragment x in A. Agent B constructs a set consisting of the conjunction of the exclusion sets (and any induced exclusion sets) for y and z . This construction gives us a kind of transitive closure. Given sufficient time, an agent can acquire knowledge about the system wide impact of executing each of its plan fragments. It does so, however, without the exchange of detailed information concerning resource availability in the system.

As before, we illustrate with the aid of our example. If agent A elects to satisfy g_1 through execution of A-b, goal g_1 cannot be satisfied unless agent B also participates. Agent B must choose either B-a or B-b. The conflict set (in B) for B-a is $\{(B-c, B-d), (B-c, B-e)\}$, so the exclusion set for B-a is $\{(g_2)\}$. Similarly, the conflict set for B-b is $\{(B-c, B-d), (B-c, B-e), (B-d, B-e)\}$, so the exclusion set for B-b is $\{(g_2) (g_4)\}$, meaning that agent B cannot participate in satisfying both goal g_2 and g_4 while at the same time participating in satisfaction of g_1 . Agent B transmits the information that $\{(g_2) (g_4)\}$ should be included in the induced exclusion set for A-b. Through this transaction, agent A learns that its choice of satisfaction of g_1 through executing A-b forces some agent elsewhere in the system to abandon either g_2 or g_4 . It does not, however, mean that g_2 and g_4 are not jointly feasible. They are simply not jointly feasible using agent B's participation, if agent A elects to execute A-b.

Though the concepts of conflict, exclusion, infeasibility, and induced exclusion have been introduced somewhat informally in this report, rigorous definitions for them have been formulated. Algorithms for determining the conflict, exclusion, and infeasibility sets associated with a plan fragment have been developed and implemented. In addition, mechanisms for transitive propagation have been devised, allowing an agent to incorporate the knowledge it acquires in its local data structures and reason using that new knowledge.

3.4.4 Multistage Negotiation

In this subsection, we describe the multistage negotiation protocol we have developed, indicating the role which reasoning about conflict and constraints plays. We first treat the protocol at a very high level, discussing the general strategy. We then provide more detail as to phases of planning and the role of negotiation in each.

Multistage negotiation provides a means by which an agent can acquire enough knowledge to reason about the impact of local activity on nonlocal state and modify its behavior accordingly. On completion of the plan generation phase, a space of alternative plans has been constructed which is distributed among the agents, with each agent only having knowledge about its local plan fragments. An agent then examines the goals it instantiated and makes a tentative commitment to the highest rated feasible set of plan fragments relative to these goals. It subsequently issues requests for confirmation of that commitment to agents who hold the contracts for completion of these plan fragments.

Each agent may receive two kinds of communications from other agents: 1) requests for confirmation of other agents' tentative commitments, and 2) responses concerning the impact of its own proposed commitments on others. Knowledge about impact of local actions is acquired through an exchange of induced exclusion and infeasibility information. The agent incorporates this new knowledge into its local induced exclusion set and infeasibility information. It rerates its own local goals using the new knowledge and possibly retracts its tentative resource commitment in order to make a more informed choice. This process of information exchange continues until a consistent set of choices can be confirmed.

Termination of the negotiation process can be done using system-wide criteria or it can be accomplished in a diffuse manner. If global termination criteria are desired in an application, some form of token passing mechanism can be used to detect that the applicable termination criteria have been met. When synchronized global termination is not required in an application, the negotiation can be terminated by an "irrevocable" commitment of resources. A node initiates plan execution in accordance with its negotiated tentative commitment at some time after it has no pending activities and no work to do for other agents.

When a node begins its planning activity, it has knowledge of a set of global goals which have been locally instantiated. A space of plans to satisfy each of these goals is formulated during plan generation without regard for any goal interaction problems. After plan generation, each node is aware of two kinds of goals: *primary goals* (or p-goals) and *secondary goals* (or s-goals). In our application, p-goals are those instantiated locally by an agent in response to an observed outage of a circuit for which the agent has primary responsibility (because the circuit terminates in the agent's subregion). These are of enhanced importance to this agent because they relate to system goals which must be satisfied by this particular agent if they are to be

satisfied at all. An agent's s-goals are those which have been instantiated as a result of a contract with some other agent. An agent regards each of its s-goals as a possible alternative to be utilized in satisfaction of some other agent's p-goal.

A plan commitment phase involving multistage negotiation is initiated next. As this phase begins, each node has knowledge about all of the p-goals and s-goals it has instantiated. Relative to each of its goals, it knows a number of alternatives for goal satisfaction. An alternative is comprised of a local plan fragment, points of interaction with other agents (relative to that plan fragment), and a measure of the cost of the alternative (to be used in making heuristic decisions). Negotiation leading to a commitment proceeds along the following lines.

- 1- Each node examines its own p-goals, making a tentative commitment to the highest rated set of locally feasible plan fragments for p-goals (s-goals are not considered at this point because some other agent has corresponding p-goals).
- 2- Each node requests that other agents attempt to confirm a plan choice consistent with its commitment. Note that an agent need only communicate with agents who can provide input relevant to this tentative commitment.
- 3- A node examines its incoming message queue for communications from other nodes. Requests for confirmation of other agents' tentative commitments are handled by adding the relevant s-goals to a set of active goals. Responses to this agent's own requests are incorporated in the local feasibility tree and used as additional knowledge in making revisions to its tentative commitment.
- 4- The set of *active goals* consists of all the local p-goals together with those s-goals that have been added (in step 3). The agent rates the alternatives associated with active goals based on their cost, any confirming evidence that the alternative is a good choice, any negative evidence in the form of nonlocal conflict information, and the importance of the goal (p-goal, s-goal, etc.). A revised tentative commitment is made to a highest rated set of locally consistent alternatives for active goals. In general, this may involve decisions to *add* plan fragments to the tentative commitment and to *delete* plan fragments from the old tentative commitment. Messages reflecting any changes in the tentative commitment and perceived conflicts with that commitment are transmitted to the appropriate agents.
- 5- The incoming message queue is examined again and activity proceeds as described above (from step 3). The process of aggregating knowledge about nonlocal conflicts continues until a node is aware of all conflicts in which its plan fragments are a contributing factor.

In the initial negotiation stage, each agent examines only its p-goals and makes a tentative commitment to a locally feasible set of plan fragments in partial satisfaction of those goals. Since each agent is considering just its p-goals at this stage, the only reason for an agent's electing not to attempt satisfaction of some top level goal

is that two or more of these goals are locally known to be infeasible. (This corresponds to an overconstrained problem.)

In subsequent stages of negotiation, both p-goals and relevant s-goals are considered in making new tentative commitments. The reasoning strategy employed at each agent will only decide to forego commitment to one of its p-goals if it has learned that satisfaction of this p-goal precludes the satisfaction of one or more other p-goals elsewhere in the system. If the system goal of satisfying all of the p-goals instantiated by agents in the network is feasible, *no agent will ever be forced to forego satisfaction of one of its p-goals* (because no agent will ever learn that its p-goal precludes others), and a desired solution will be found. If, on the other hand, the problem is overconstrained, some set of p-goals cannot be satisfied and the system tries to satisfy as many as it can. While there is no guarantee of optimality, the heuristics employed should ensure that a reasonably thorough search is made. The propagation mechanisms associated with induced exclusion ensure that (given sufficient time), each agent could gain complete knowledge about the nonlocal impact of its own local alternatives.

3.5 Distributed Knowledge Base Management

The problem solving system we are building requires a broad range of diverse kinds of knowledge. The knowledge base differs in many respects from that found in a typical "expert system". Unlike most expert systems, we are concerned with solving *multiple types* of problems in a *geographically distributed* environment. Our application domain involves a large communications network partitioned into subregions with one site in each subregion functioning as a central controller over its local subregion. From a knowledge base perspective, we are interested in problem solving paradigms which lead to effective cooperation among the subregional control centers in the absence of global knowledge about the network. In designing the architectural framework for this problem solving system we also identified a requirement for sharing certain domain specific knowledge among different agents within a single subregional control center. This section discusses our work on the design of a distributed knowledge base manager (KBM).

3.5.1 Introduction

Although our system incorporates a range of knowledge types, we restrict our attention in this section to the body of knowledge associated with network structure, the various types of network components and their interconnections. This structural knowledge about the network is initially acquired as a single global knowledge base using a specially developed tool, called GUS, which provides a convenient graphical user interface.

GUS constructs a domain-specific centralized knowledge base by allowing the user to instantiate objects in predefined classes and create allowable relations between any two or more objects. GUS enforces the domain-specific rules governing object-object relations. In our case, the typical classes are subregions, sites, radios, multiplexers, trunks, and supergroups. Rules define how a radio may be connected to a multiplexer, how many circuits may be assigned to a trunk, etc. After all of the objects and their interrelations have been created, the user may select an option which breaks the single knowledge base into several local knowledge bases. Each local knowledge base contains the information we would expect a controller at a subregion control center to know.

The Knowledge Base Manager (KBM) is a distributed collection of software modules (one for each subregion control center) which have the responsibility of managing this large, distributed and shared knowledge base. During this year we have started the initial design and implementation of the KBM. Several important research issues have arisen and been investigated. In the next subsections, we discuss these areas, describe our progress thus far, and conclude with a view toward future work.

3.5.2 Representational Issues in Domain Knowledge

To understand the representational issues associated with the knowledge structure

for circuits, two primary terms in telecommunications service -- trunk and circuit -- must be explained. A circuit in our domain is the complete elementary path between two pieces of terminal equipment by which a two-way telecommunications service is provided [6]. In other words, a circuit represents the notion that two people are talking together, or two machines are exchanging data. A trunk is a group of equipment and connections which establishes, at a higher level of abstraction, telecommunications connectivity by providing a resource for circuits to follow. Circuits ride on channels of a trunk; there is typically a capacity for several channels per trunk. A useful analogy for channels on a trunk is to imagine one big pipe (the trunk), which contains a number of smaller pipes (channels) running the entire length of the big pipe. With this analogy it is easy to understand why a trunk can ride channels of other trunks. The trunk exists with or without the circuit, but this is not symmetric; a circuit cannot exist unless it rides a trunk, or a list of trunks connected in series. The trunk refers to "physical" connectivity, and the circuit refers to "logical" connectivity.

Although there was a clear distinction between physical and logical connectivity in our initial design, we did not understand the need to separate the two concepts and combined both into a single knowledge structure for circuit. The frame for a particular circuit included certain physical information, such as the equipment endpoints and the input numbers of the equipment endpoints in a multiplexing scheme, and certain logical information, such as the restoration priority and status. Note that the physical information in a circuit instance can be used to derive the trunk which it rides, but the trunk is not explicitly defined. In fact, the set of trunks which are present in a sample communications network were not explicitly defined anywhere in the knowledge base.

Upon further development of our distributed problem solving environment, we found that certain problem solvers must base much of their reasoning activities in our domain on physical connectivity at a higher level of abstraction than equipment and connections. This higher level of abstraction involves the set of resources needed to carry a circuit. A trunk channel is the basic unit of this resource. For example, if a circuit has been disrupted, then the service restoral planner will attempt to restore the circuit by routing it on alternative trunks. That is, the logical connection between two users will be reestablished by choosing a different series of equipment and connections. While it is possible to search for these equipment objects and connection objects, it is much more effective to deal with them on the trunk level.

As a result of these observations, we decided to redesign portions of the knowledge base to include trunks. This involved creating a new knowledge structure for circuit. Our old concept of circuit which included both physical and logical knowledge is now partitioned into trunks (physical) and circuits (logical). The trunk knowledge structure includes the physical path of a series of equipment and connections. This path level knowledge groups subsets of equipment and connections together, a grouping which, as described above, is necessary in our problem solving environment. Circuits are purely a logical entity now. A circuit instance includes the trunk or list of trunks which it currently rides, along with the channel numbers of those trunks. Only trunk objects contain knowledge about physical connectivity.

Although the primary reason for redefining the circuit knowledge representation was the need for problem solvers to reason about groups of objects, an additional advantage of this approach is the enhancement introduced to GUS as a result of implementing trunks and restructuring circuit knowledge. GUS is more powerful and user-friendly now because it is capable of representing and displaying knowledge at a higher level of abstraction to the user. The purpose of the tool is to provide a mechanism for knowledge acquisition and representation in the most "natural" setting possible. Clearly, humans do not represent their knowledge on simply one level, such as equipment and connections. The hierarchical levels of abstraction are used by experts and should be included in any knowledge acquisition tool. In our domain, trunks give GUS the ability to inform the user when an attempt has been made to include knowledge which is incomplete or incorrect at a higher level of abstraction. For example, if the user attempts to create a circuit without having previously created the necessary trunk, GUS recognizes the problem at two levels. At the higher level of abstraction, a circuit cannot be created if there are no resources to carry it. At the lower level, a piece of equipment or connection is missing or configured improperly. GUS is thus more powerful because it can interact with the user on more than one level. The real significance of this is that the user is more likely to understand the nature of the difficulty when he is presented with more than one description of a problem.

3.5.3 Distribution of Global Network Knowledge

In the design of a distributed problem-solving system, non-trivial problems may arise when distributing all forms of knowledge contained within centralized (global) knowledge structures among a set of distributed (local) knowledge bases. Distributed knowledge bases are similar to conventional distributed databases in that there are different ways in which the knowledge can be distributed. These various ways may be classified as partitioned, replicated, or hybrid.

To understand each model, it is convenient to think of the centralized knowledge base as a file cabinet containing a number of folders. To create the distributed knowledge base, the folders contained within the file cabinet must be moved to a set of briefcases. In the partitioned model, each file is removed from the cabinet and placed in a briefcase. Thus, each individual piece of knowledge (assuming the naive notion that all knowledge is internally independent) contained within the centralized knowledge base now exists in the distributed environment in only one of the local knowledge bases. In the replicated model, the basic idea is that the folders are removed, copied and then placed in a number of briefcases. That is, one piece of knowledge from the centralized knowledge base exists at more than one local knowledge base. The hybrid model contains some knowledge which is partitioned and some which is replicated. Our design is based on a hybrid model of a distributed knowledge base.

Our domain specifically defines much of the manner in which we distribute the central knowledge base into local knowledge bases. We divide the knowledge along subregion boundaries because this represents an accurate model of real-world activities. In the communications network, as in our distributed knowledge base model,

much of the centralized knowledge is strictly replicated or partitioned, but there is also a need for hybrid knowledge representation.

The two types of knowledge which are replicated in each local knowledge base are knowledge about generic objects and network-specific knowledge. Examples of generic objects at the equipment and connection level include radios, digroups, supergroups, and multiplexers. Information within each local knowledge base pertaining to a generic radio, for instance, includes how it can be connected to other generic objects, what alarms may exist and what each alarm means. Also included is knowledge about a generic site, link, and subregion. This type of knowledge is necessary so that knowledge base managers can communicate with one another. For example, when one knowledge base manager asks another about a particular link which they have in common, each manager must be able to understand the fundamental properties of a generic link in order to communicate effectively. This cooperation among the knowledge base managers also requires network-specific knowledge. The most important piece of knowledge in this category is a global connectivity map, which defines the overall network layout in very general terms -- names of existing subregions and their interconnections. This type of system level knowledge is necessary to process many knowledge base queries.

Knowledge about instances of class objects are generally partitioned, and thus are included in only one of the local knowledge bases. For example, all knowledge about an equipment instance is included only in the local knowledge base for the subregion in which it is located. Knowledge about connection instances are generally the same, with one exception which will be discussed below. In general, equipment and connection knowledge is partitioned because we do not expect a problem solving agent in one subregion to know about the details of equipment and connections in another subregion.

In our domain, some knowledge must be present within two or more subregions but not in every subregion and is thus distributed in a hybrid model. These particular subregions all have knowledge of different aspects of an object in the centralized knowledge base. This kind of knowledge can be described as pertaining to an object which in some way crosses subregion boundaries. Examples are trunks, circuits, and certain links. A link conceptually connects two radios at different sites. Because it is a type of connection, and knowledge about the equipment endpoints of connections is included within the local knowledge base in which the connection is contained, knowledge about the radios at the ends of a link is included within the knowledge base in which the link is included. This leads to the exception mentioned in the previous paragraph. When a link connects radios in two sites which are not within the same subregion, each subregion does not contain knowledge about the radio at the far end. Each subregion only knows the site name and the other subregion to which the link connects. Because each local knowledge base contains information, but not exactly the same information, about this link, this knowledge requires a hybrid model. Similarly, each subregion controller must know about the priority, status, and local trunk route of every circuit which uses a local resource within that subregion. This is needed when attempting to reallocate resources (trunks) in order to restore circuits. Each subregion must also have knowledge about the trunks which are local resources available for the service restoral planner in restoring a circuit.

3.5.4 Management of a Distributed Knowledge Base

Managing a distributed knowledge base combines many of the features of centralized artificial intelligence knowledge bases with classic distributed databases. From the artificial intelligence perspective, there should be, for example, control structures for deductive, plausible, and inductive reasoning, techniques for knowledge acquisition, refinement, and validation, as well as uncertainty management. Database management systems contribute such aspects as semantic data models, concurrency control, error recovery, and query processing. Note, however, that these characteristics are interrelated and cannot always be clearly differentiated.

To understand the complexity of managing a distributed knowledge base, we extend the knowledge base analogy presented in the previous section, where similarities were drawn between distributing a central knowledge base into a set of local knowledge bases and removing data folders from a file cabinet to be placed in a number of briefcases. The purpose is to examine the differences between managing conventional distributed data bases and the techniques for distributed knowledge base management.

If the knowledge base is not shared and not distributed, then it is managed by its user. This is the case in most expert systems, in which there is no specific function for knowledge base management. The need for management comes from the desire to share a common knowledge base among several problem solving agents. Continuing the analogy from above, this situation is modeled by having one person in charge of managing the knowledge. That is, one person sits by the file cabinet and uses the knowledge contained within the cabinet to answer questions from others. While he is not answering queries, he does other routine bookkeeping such as updating the files, removing old ones, and perhaps dealing with files which contradict each other.

After the files have been placed in separate briefcases and moved to different parts of the country, a person is assigned to each briefcase to manage its contents. Collectively, these people perform the tasks of the one person who had previously sat by the file cabinet. Now, because the knowledge is distributed, each person must know how to perform the local job of knowledge base management, as well as knowing how to cooperate with the other managers. For example, if one of these managers were questioned about something in the knowledge base that is not known locally, he should be able to call upon the appropriate person — someone who is managing another briefcase — to ask if that person is able to help.

The overall knowledge base structure as a result of moving the files to briefcases is very similar to the knowledge base in the domain of our distributed problem solving environment. Within a local knowledge base is a global connectivity map at the level of subregions and connections between subregions. By interpreting that information, a knowledge base manager can determine which other knowledge base managers exist. "Addresses" needed to communicate with the others are also included. In performing its tasks of managing the knowledge, a knowledge base manager may need to consult other knowledge base managers. It does this by sending a message to the other knowledge base

manager.

Two functions of the knowledge base management system we have been most interested in designing and implementing are a programmer's interface to the local knowledge base and an interface between the local knowledge base and the service restoral agent. Because the latter has been developed to the point that we can run simulations of the circuit restoral process, we have been able to make significant improvements in the implementation of multistage negotiation. Each of these two features of knowledge base management will be discussed in detail.

3.5.4.1 Natural Language Interface

During the execution of problem solving software, it is often difficult to determine the effectiveness and correctness of the knowledge base management system. SIMULACT provides only limited screen space for each actor, so a knowledge base manager cannot continually inform the user about its activity using screen displays. Many of the tests we conduct proceed rather quickly, and thus even if each knowledge base manager could write effectively to the screen, it would require interrupting each of these simulated parallel processes in order to provide effective user interaction. In addition, it is inherently difficult for humans to assess parallel, distributed processes.

In order to meet the need for user interaction with a local knowledge base, we have developed an interface which will accept questions and provide replies based on the content of the local knowledge base. This interface uses simple natural language techniques so that it appears to understand very free form queries. The user need not be familiar with the syntax and semantics of the knowledge base. We are pursuing this type of interface because of its ease of use and eventual ability to give explanations in English sentences.

The natural language interface is currently being used to determine the state of the network at various times in the simulation. This subsystem of a local knowledge base manager can answer questions about the state of a particular instance of a class of objects by directly accessing its knowledge structure. It can also answer queries which involve reasoning rather than simple look-up such as how might it reroute a particular circuit based upon the current state of the local knowledge base. In the future, the system might be developed to answer queries based upon information which is not currently true. A query of this form might be "If link 1 failed, how would you reroute circuit 5?" Also on the horizon are the explanation facilities. Currently, only the framework for this has been created.

3.5.4.2 Interface with Planning

One task of the knowledge base manager is that of interfacing with problem solving agents which require access to the knowledge. The knowledge base manager must be able to accept queries and process them in an order which is logically correct. In addition,

responses must be in a form which is understandable by the agent who requested the knowledge. Answering a query typically involves more than simple information look-up, instead requiring some degree of reasoning.

We have focused the development of this aspect of knowledge base management on the interface with the service restoral planner which is responsible for restoring disrupted circuits. In general, restoring a circuit involves cooperation among several service restoral agents. The basic approach involves a distributed search through all possible subregions which could have resources available to be used for restoral of the circuit.

The procedure for rerouting a circuit is fairly complex. When a circuit becomes disrupted, the service restoral agent for each subregion in which the circuit terminates is notified that the circuit must be restored. To illustrate the complexity of the problem we will discuss a circuit which has endpoints in two different subregions, and which passes through at least one other subregion. In this case, each of the two service restoral agents at either end of the circuit initiates, in parallel, a request of its local knowledge base manager for a list of available resources which could be used to reroute the circuit locally. The response is a list of combinations of trunks and their respective channels. The last trunk in each sequence connects to another subregion; thus, the service restoral agent must then cooperate with the service restoral agent in the subregion at the trunk terminus to further the circuit toward the distant endpoint. Upon completion of this search process for all possible circuit restoral plans, the service restoral agents negotiate to select a single restoral plan for each circuit.

The queries from the service restoral agent for the knowledge base manager fall into two categories: primary and secondary. A primary query is associated with a primary subgoal and seeks resources to reroute a circuit which originates within the subregion. A secondary subgoal generates a secondary query for resources to reroute a circuit coming into the subregion using a specific shared resource. For a primary query, the service restoral agent simply knows the name or ID of the circuit; all knowledge pertaining to that circuit is kept within the knowledge base. For a secondary query, the service restoral agent provides the ID of the circuit, its priority, and the channel of the particular trunk used by the circuit to enter the subregion. These attributes were passed from the service restoral agent which generated the request for the secondary subgoal. In either case, the knowledge base manager must respond by providing a list of possible resources, if any are available, which could be allocated by the service restoral agent to restore the circuit.

An important differentiation of function has been made here. The task of the service restoral agents is to generate global plans based on resource availability and to select plans based on negotiations over resource allocation. The knowledge base manager serves as a bridge between the planning specific knowledge of the service restoral agents and the domain specific knowledge about resources contained within the the knowledge base.

3.5.5 Future Work

The current development of the knowledge base manager has concentrated on query processing. As we continue this work by adding interfaces with other problem solving activities, we intend to begin implementing and testing other aspects of knowledge base management. One of these aspects is belief revision or truth maintenance. Truth maintenance is used to control the validity of a knowledge base, particularly when new information is introduced.

Knowledge base construction begins with the creation of a set of hypotheses upon which new conclusions are based. In this process inconsistencies may arise which, for instance, might imply a particular statement to be true and also not to be true. Truth maintenance is used to logically backtrack through the knowledge to determine the source of this inconsistency.

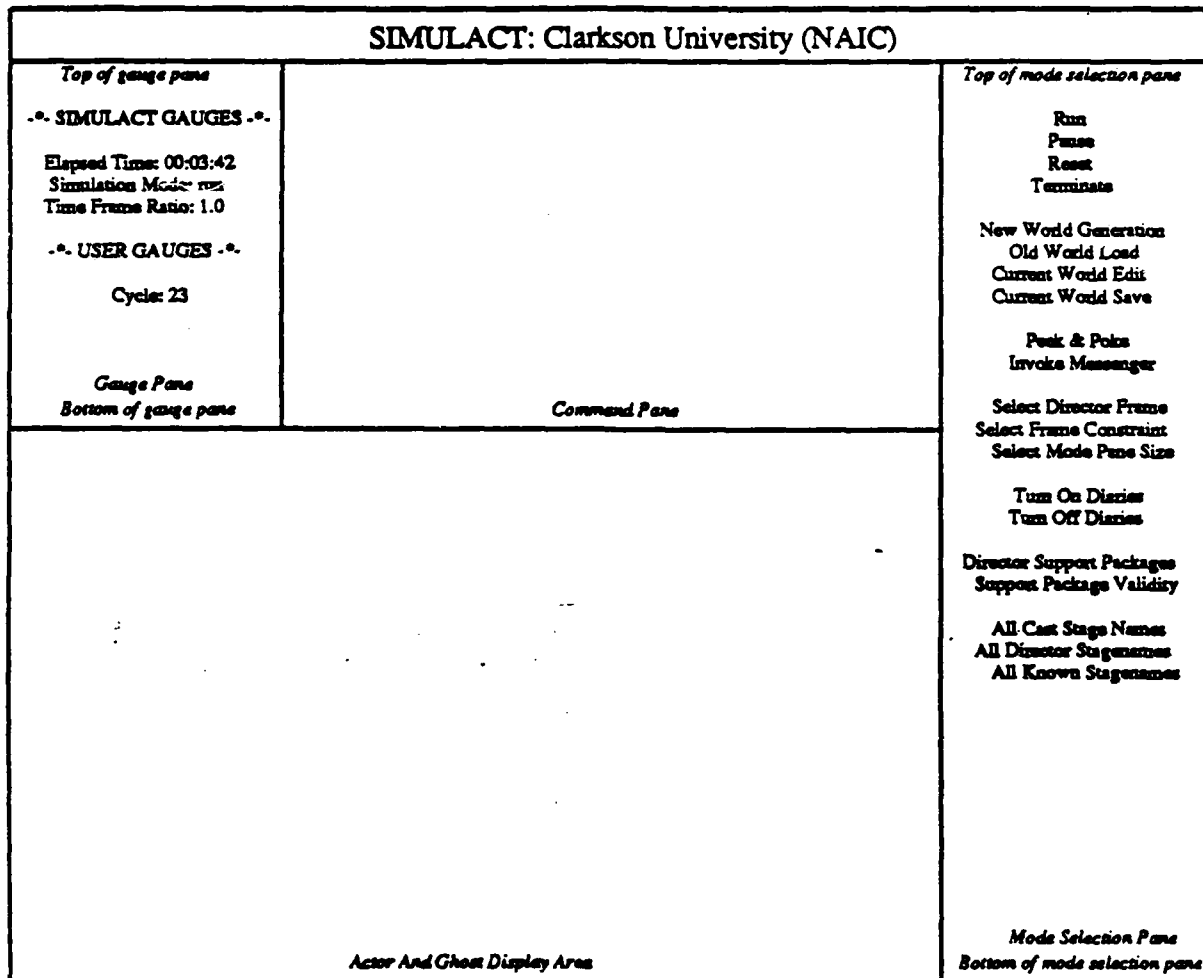
Performing truth maintenance in our distributed environment is particularly interesting because a local knowledge base certainly should be consistent and without contradictions. However, it may not be necessary for the knowledge in the system as a whole to be perfectly consistent. At this point in our work it is too early to determine just how "globally consistent" our knowledge must be. We are interested in cooperative, distributed problem solving; there are few applications in the real world in which there are no contradictions in global knowledge. For example, in any group of experts working together on a problem, there are likely to be inconsistencies in the collective set of their knowledge. Yet they are usually able to work successfully toward a mutually satisfying solution. A fundamental problem for distributed knowledge based systems is to determine the body of knowledge which should be consistent and that which need not be in order for the system to perform effectively.

Bibliography

1. Meyer, Robert A. and Meyer, Charles, "The Role of Knowledge-based Systems in Communications System Control," *Applications of Artificial Intelligence V, Proc. of SPIE*, vol. 786, 18-20 May 1987, pp. 305-310.
2. Adams, Gerald M., Meyer, Charles N. and Meyer, Robert A., "Machine Intelligence For DoD Communications System Control," *Conf. Record MILCOM-87*, vol. 1, 19-22 October 1987, pp. 188-193.
3. MacIntosh, Douglas J. and Conry, Susan E., "A Distributed Development Environment for Distributed Systems," *Proc. Third Annual Expert Systems in Government Conference*, 19-23 October 1987, pp. 72-79.
4. MacIntosh, D. J. and Conry, S. E., "SIMULACT, A Generic Tool for Simulating Distributed Systems," *Tools for the Simulation Profession*, The Society for Computer Simulation. San Diego, CA, April 1987, pp. 18-23.
5. Hogencamp, Brian R., *GUS: A Graphical User Interface for Capturing Structural Knowledge*, M.S. Thesis, Clarkson University. Potsdam, NY, January 1987.
6. *Baseline Conceptual Design for the DCOSS Data Base*, Defense Communications Agency, July 1985.

SIMULACT USER'S MANUAL

(Draft December 1987)



By Douglas J. Mac Intosh

1. Overview of SIMULACT

SIMULACT is an environment intended to aid the development of applications involving distributed problem solving. It is a domain independent development and emulation facility which permits rapid prototyping, interactive experimentation, and ease of modification of such systems.

SIMULACT is based on a model which regards intelligent agents as semi-autonomous problem solving agents which interact with one another by means of a message passing paradigm. This model assumes no shared memory between agents. SIMULACT is a distributed system that allows n agents to be modelled on k machines where $n > k$. This system is written in Common Lisp and can run on a non-homogeneous network of Lisp Machines comprised of both SYMBOLICS (version 7.1) and TI EXPLORERS (version 3.1).

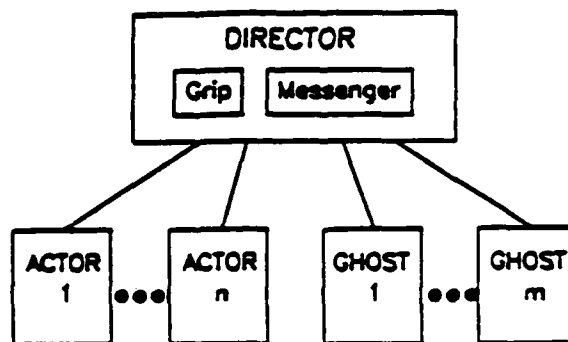
1.1 System Structure

SIMULACT is a distributed system that allows n agents to be modelled on k machines, where $n > k$. Each agent runs asynchronously and coordinates its activity with that of other agents through the exchange of messages. The activities performed by each agent are assumed to be complex, so that the parallelism is coarse grained. SIMULACT allows the programmer to write code in Common Lisp as though there were as many Lisp machines in the network as there are agents in the distributed system being developed.

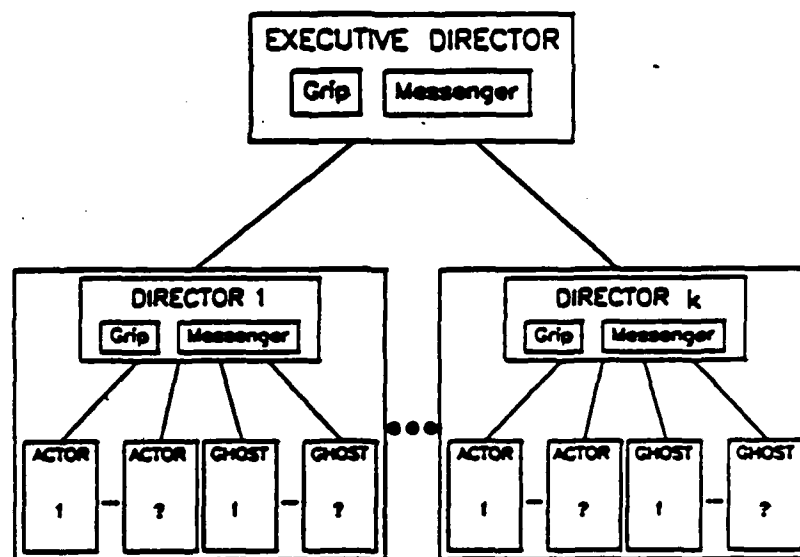
As is evident from Figure 1, SIMULACT is comprised of four component types: Actors, Ghosts, Directors, and an Executive Director. Actors are used to model agents in the distributed environment. Each Actor type is individually defined, and used as a template to create multiple instances of that Actor type. An Actor is a self contained process which runs in its own non-shared local environment. Although Actors run asynchronously, the elapsed CPU time for each actor never varies by more than one "time frame".

Ghosts are used in SIMULACT to generate and inject information into the model that would naturally occur in a "real" distributed expert system. They do not represent any physical component of the model. For example, external inputs (alarms, sensors, etc.) affecting the state of the system can be introduced via Ghosts, as well as inputs that reflect the "side effects" of the systems activities. Ghosts can also be used to inject noise or erroneous information into the system so that issues concerning robustness can be easily investigated. The performance of an expert system can be monitored in subsequent runs through the simple modification of these Ghosts.

Due to the similarities between Actors and Ghosts, we refer to them as Cast members. Each Cast member has a unique "stagename" and a "mailbox" used by the the communication facility in routing messages among members. Each also has a "script function" which defines its high level activity.



a) Host Level Structure



b) Network Level Structure

Figure 1. SIMULACT's Modular Structure

The control structure residing at each host processor in SIMULACT's distributed environment is known as the Director. The Director is responsible for controlling the

activities of the Cast members at that site, and for routing messages to and from these members. These activities are assigned to the Grip and Messenger respectively. The responsibilities of the Grip range from setting up and initializing each Cast member's local environment to managing and executing the Actor and Ghost queues. The Messenger only deals with the delivery and routing of messages. When a message is sent, it is placed directly into the Messenger's "message-center". During each time frame, the Grip invokes the Messenger to distribute the messages. Whenever the destination stagename is known to the Messenger, the message is placed in the appropriate Cast member's mailbox. Otherwise, it is passed to the Executive Director's Messenger and routed to the appropriate Host.

There is one Executive Director in SIMULACT which coordinates all Cast member activities over an entire network. The Executive Director provides the link between Directors necessary for inter-machine communications, directs each Grip so that synchronization throughout the network is maintained, and handles the interface between the user and SIMULACT.

1.2 Concurrency Control in SIMULACT

Concurrent execution of n Actors on k machines ($n > k$) is emulated through the imposition of a "time frame" structure in execution. A time frame cycle breaks down to three fundamental parts: invocation of the Ghosts, the distribution of mail by the Messengers, and invocation of the Actors. For SIMULACT distributed over two hosts, Figure 2 depicts a representation of two time frames.

At the start of the first time frame, the Executive Director notifies both Directors to begin executing Ghosts. (This models the occurrence of events in the world external to the distributed system.) At the conclusion of the Ghost frame, each Director automatically invokes its Messenger. The Messenger distributes all messages which were generated during the current Ghost frame, as well as all those resulting from the previous Actor frame. Mail destined for Cast members residing on the same host processor is placed in the appropriate mailboxes. The solid line extending from each Director's Messenger represents the transfer of non-local mail to the Executive Director's Messenger. In order to reduce network overhead, this transfer is done in the form of a single message. This communication always occurs, even if there are no messages to distribute, as a synchronizing mechanism for the time frame so that Actors cannot "run away". After sending this message, each Director enters a wait state until the Actor frame directive is received from the Executive Director. The dotted line directed out of the Executive Director's Messenger represents the possible distribution of inter-machine mail prior to sending the Actor frame instruction. The Executive Director's Messenger is invoked immediately following the receipt of the last Director's Messenger communication.

Upon receiving an Actor frame command from the Executive Director, the Director's Messenger is invoked to distribute any inter-machine messages that may have been received. Next, each Actor is allowed to run for one time slice (time frame). At this

point the Executive Director immediately enters its next time frame cycle, sends the Ghost frame command, and waits for all the Director Messengers to send their next synchronizing signal. Again in the second time frame of Figure 2, it is Director 2 which requires the most time to run.

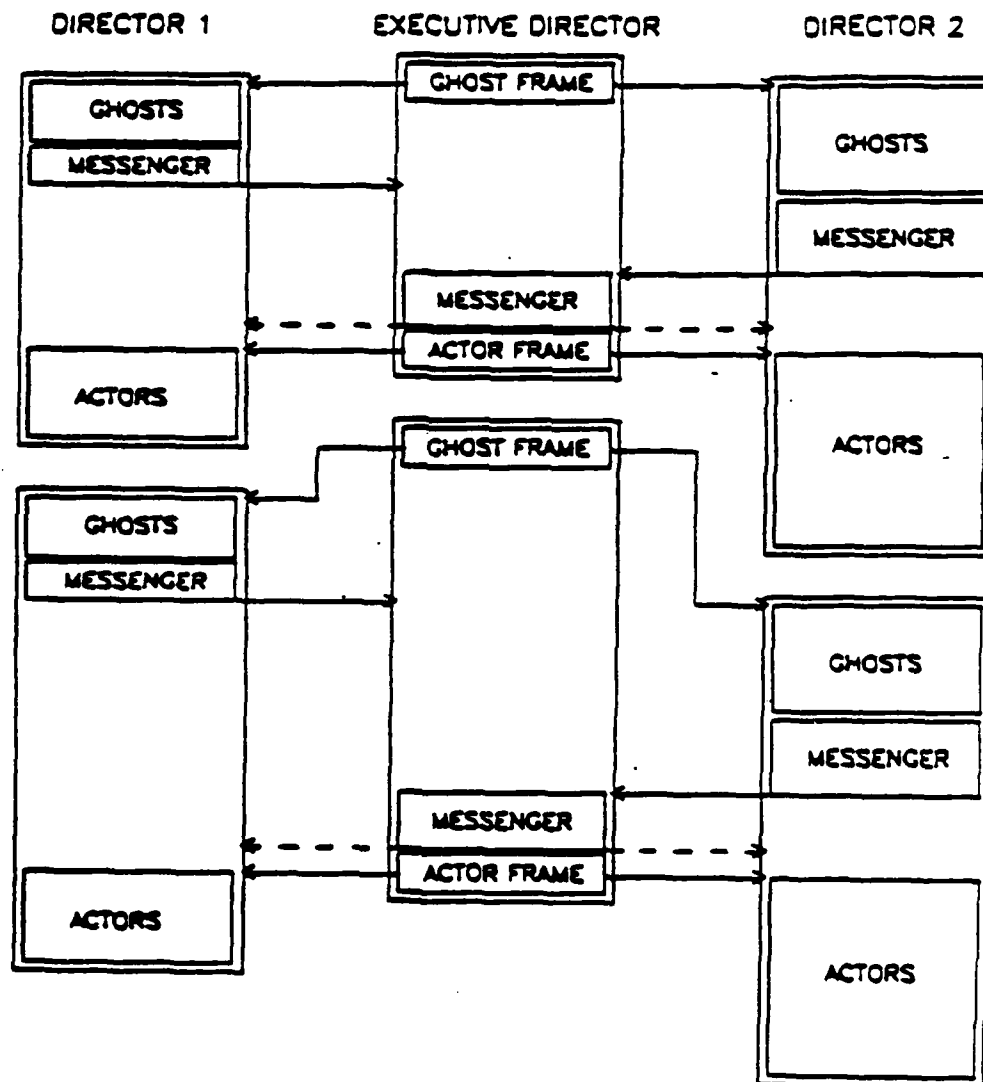


Figure 2. SIMULACT's Time Frame

2. Getting Started in SIMULACT

The SIMULACT environment is constructed from both domain independent and domain dependent code. The domain independent code is comprised of system code that is used to create the basic shell. The user then customizes the environment by loading application code. Since SIMULACT is a developmental tool, it can be expected that application code will be revised on a regular basis. For these reasons, SIMULACT's system code is typically placed in its own directory. Furthermore, it is recommended that each user developing the system keeps their application code in separate directories.

2.1 Loading the SIMULACT Distribution Tape

SIMULACT is a distributed environment capable of running on a non-homogeneous network of Lisp machines. The distribution tape contains software capable of running on both SYMBOLICS (version 7.1) and TI EXPLORERS (version 3.1). Depending on your own network configuration, you will need to select one machine to download the distribution tape to. Typically this machine is the file server for the network. Refer to section 2.1.1 for a SYMBOLICS host or section 2.1.2 for a TI EXPLORER host.

2.1.1 Down Loading to a SYMBOLICS

Down loading SIMULACT to the SYMBOLICS is a two step process; the first step requires you to read in the contents of the distribution tape, while the second step creates your own SIMULACT system. Before loading the distribution tape, you need to decide where to place SIMULACT's system code. Typically this is placed in the top level directory ">SIMULACT>".

To load SIMULACT from the distribution tape, type the following command into the Lisp Listener Window:

Command: (tape:carry-load)

You will see something similar to:

```
Carry Dump made by DOUG.  
Dump taken at 12/21/87 14:55:22.  
Dumped on machine Blackfriars.  
Dumped: Blackfriars:>SIMULACT>*.*.Newest.  
Load Blackfriars:>SIMULACT>some-file into YOUR-HOST:>YOU>some-file (Y, N, O, or A)
```

At this point type O at the keyboard to select "Other". Now enter your chosen directory pathname for SIMULACT and press Return. When loading the next file a second "load" query will appear. At this time select option "All" by typing A at the keyboard. The remaining files on the carry tape will now be loaded without any further queries.

At this point you are now ready to create the SIMULACT system(s). See section 2.2

"Creating The SIMULACT System".

2.1.2 Down Loading to a TI EXPLORER

Down loading SIMULACT to a TI EXPLORER is a two step process; the first step requires you to read in the contents of the distribution tape, while the second step creates your own SIMULACT system. Before loading the distribution tape, you need to decide where to place SIMULACT's sytem code. Typically this is placed in the top level directory "SIMULACT;".

To load SIMULACT from the distribution tape, enter the EXPLORER's Backup System by pressing select-B. Place the distribution tape in the tape drive and complete the following steps:

1. Prepare the tape by mousing on the "Prepare Tape" menu item. Now select the "Carry Tape Format".
2. Load the tape by mousing on the "Load Tape" menu item.
3. Now select the "Load Carry Tape" menu item. At this point you will need to specify where to load the contents of the tape. Enter your chosen directory pathname.

When the "Load Carry Tape" option has completed, you are now ready to create the SIMULACT system(s). See section 2.2 "Creating The SIMULACT System".

2.2 Creating The SIMULACT System

Both SYMBOLICS and TI EXPLORERS allow large software projects to be defined as "systems". If the operator installing SIMULACT is not familiar with the system concept they should refer to their Lisp machine's manual (SYMBOLICS - Manual 4, "Defining A System". TI EXPLORER - Programming Concepts Manual, "defsystem and make-system").

A SIMULACT system is generated automatically by the function `create-a-simulact-system`. In order to invoke this function you must first load file "MAKE-SIM" from the >SIMULACT> directory.

`create-a-simulact-system` *name application-directory* *Function*
Execution of this function creates a SIMULACT system called *name*. The `defsystem` files "*name.lisp*", "*name.system*", and "*name.translarion*" are automatically created and stored by this function. `create-simulact-system` is sensitive to whether you have a SYMBOLICS or TI EXPLORER environment and will tailor your system files appropriately.

The *application-directory* argument specifies the default directory to be used when accessing application code.

2.3 A Demonstration of Your SIMULACT Environment

After completing the previous steps you are ready to bring up and test your SIMULACT environment. The first thing you need to do is to make the SIMULACT system. The command to do this for a SYMBOLICS is:

Load System "*name*"

where *name* is the name you called the system when creating it. For a TI EXPLORER execute the following:

(make-system "*name*" :nowarn)

Again, *name* is what you called the SIMULACT system when creating it.

You are now ready to run SIMULACT. Type (SIMULACT) in the Lisp Listener and momentarily the SIMULACT's main window frame will appear (see figure 3). Shortly thereafter the *SIMULACT Initialization* window will appear in the command pane. Mouse on "Old World Load" to bring up a menu of *SIMULACT Worlds*. Now use the mouse to select the demo you would like to view. After the demo world is loaded in, mouse on "Run" in the *Mode Select Pane* to start the demo.

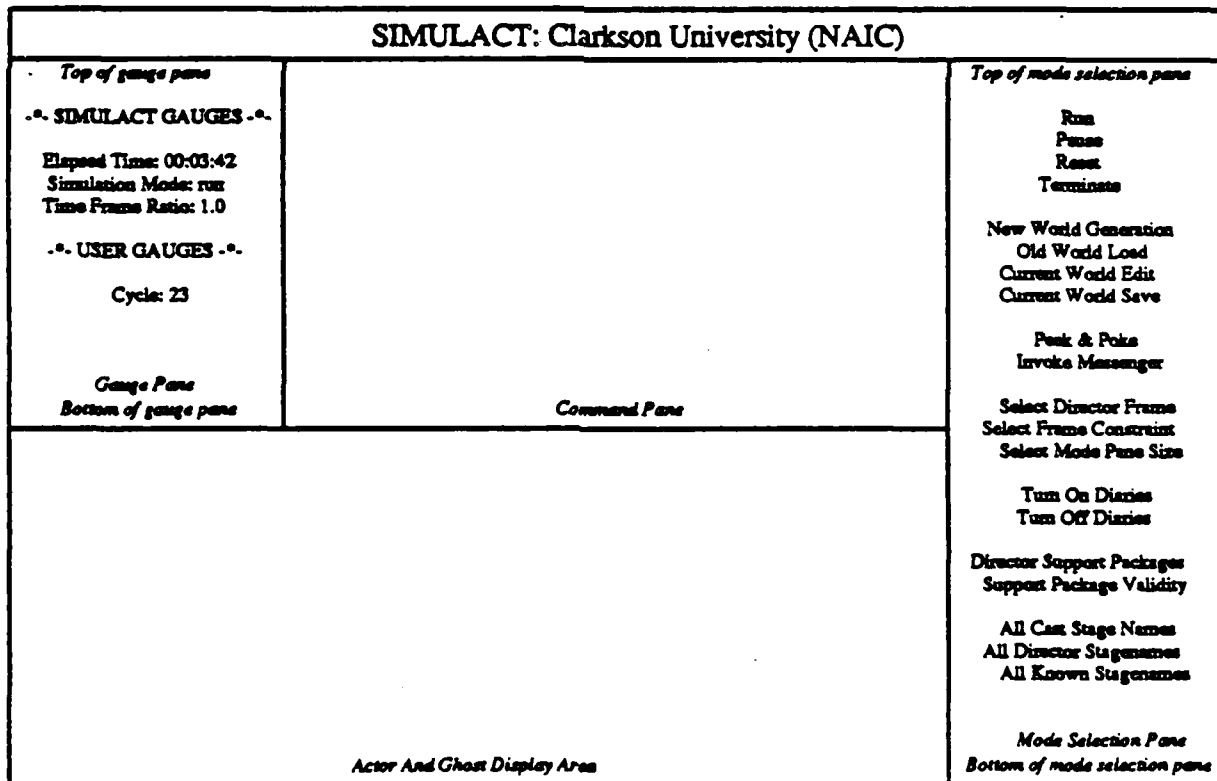


Figure 3. SIMULACT's Main Window Frame

3. User Interface Facilities

3.1 The Mail Facility

The Mail facility provides the programmer with three mechanisms for communication among Cast members: *memos*, *futures*, and *future streams*. In general, a *packet* of information is sent from one Cast member to another, addressing the target member by its *stagename*. The format of these packets is not specified by SIMULACT. It is left up to the user to formulate a syntax that is convenient in the context of the system being developed. A packet may be any *symbolic Lisp form* that can be evaluated in the targetted Cast member's environment.

A Cast member has a *mailbox* and a *futurebox* used to collect incoming memos and futures. It is the responsibility of the programmer to periodically check these boxes for mail using the *memos-p* and *futures-p* functions.

memos-p

Function

memos-p returns *t* if one or more memos are present in the Cast member's mailbox, otherwise *nil*.

futures-p

Function

futures-p returns *t* if one or more futures are present in the Cast member's futurebox, otherwise *nil*.

3.1.1 Memos

Sending a memo is the simplest mechanism one Cast member can use to communicate with another. It is a "one way" transfer of information that automatically appears in the destination Cast member's mailbox at the beginning of the next time frame. Memos are sent and retrieved by a Cast member using the *send-memo* and *receive-memos* functions respectively.

send-memo destination memo

Function

Creates, initializes, and sends an instance of the *sim-io:memo* flavor to the Cast member whose *stagename* is *destination*. *destination* must be a Common Lisp string. *memo* may be any symbolic Lisp form that can be evaluated in *destination's* local environment. The *source* and *memo-time* instance variables of *sim-io:memo* are automatically set to the *stagename* and current elapsed time of the sending Cast member. *send-memo* returns the created object.

If both *source* and *destination* Cast members reside on the same Lisp machine, they share one instance of the *sim-io:memo* flavor. In order not to violate the underlying assumption that each Cast member has an independent environment, the instance variables of *sim-io:memo* are not settable.

receive-memos

Function

Returns and clears the contents of a Cast member's mailbox. If mail is present

this function returns a list of memos, otherwise nil. A memo is an instance of the `sim-io:memo` flavor.

The extraction of *memo content* can be done in a number of various ways. The following are methods of the `sim-io:memo` flavor available to the programmer for this purpose. Functions that perform similar operations to those of these methods, but may be applied to a more general *mail-object* (memo, future, or future stream) are also introduced.

:list of sim-io:memo *Method*
Returns a list of four elements, (*destination source memo-time memo*).

:destination of sim-io:memo *Method*
Returns the destination Cast member's stagename.

destination mail-object *Function*
Returns the value of *mail-object's destination* instance variable. *mail-object* may be an instance of `sim-io:memo` or `sim-io:future`.

:source of sim-io:memo *Method*
Returns sending Cast member's stagename.

source mail-object *Function*
Returns the value of *mail-object's source* instance variable. *mail-object* may be an instance of `sim-io:memo` or `sim-io:future`.

:memo-time of sim-io:memo *Method*
Returns the the time that this memo was sent.

memo-time mail-object *Function*
Returns the time *mail-object* was sent. *mail-object* may be an instance of `sim-io:memo` or `sim-io:future`.

:memo of sim-io:memo *Method*
Returns the value of the *memo* instance variable. This is the packet of information transferred from the *source* Cast member to the *destination* member.

memo mail-object *Function*
Returns the value of the memo instance variable of *mail-object*. This is the packet of information sent from the *source* Cast member to the *destination* member. *mail-object* may be an instance of `sim-io:memo` or `sim-io:future`.

3.1.2 Futures And Future Streams

Many communications between Cast members take the form of requests for information. Using the `send-memo` function requires that the sending member sort its mailbox to retrieve the reply to a request after it has been received. *Futures* provide a Cast member with a direct mechanism for sending a message that returns a result. A memo sent

using the *send-future* function returns an instance of the *sim-io:future* flavor. After the memo has been received and processed, the result is transparently routed back to this object by *SIMULACT*. The sending Cast member uses this future to determine when the result is available, and to extract it after it has arrived.

send-future *destination memo*

Function

Creates, initializes, and returns an instance of the *sim-io:future* flavor. The *source* and *memo-time* instance variables are automatically set to the stagename and current elapsed time of the sending member. A second instance, and duplicate copy of this future is placed in *destination's* futurebox at the beginning of the next time frame. This object is held on to, and periodically tested by the sending Cast member to determine when a reply to *memo* is received. When available, it then can be extracted from this object.

In some cases, requests for information may not have one definitive reply. Instead, pieces of information may be returned at different times. *SIMULACT* allows two Cast members to establish a *future stream* between themselves for returning results over time. In this case, the *send-future-stream* function is used instead of *send-future*.

send-future-stream *destination memo*

Function

Creates, initializes, and returns an instance of the *sim-io:future* flavor. This instance has its *source* and *memo-time* instance variables set to the stagename and current elapsed time of the sending member. At the beginning of the next time frame, a second instance and duplicate copy of this future stream is placed in *destination's* futurebox. This object is retained, and periodically tested by the sending Cast member to determine when one or more replies to *memo* have been received. These replies are then extracted one at a time from the object.

Sending a future or future stream uses two instances of the *sim-io:future* flavor. The sending Cast member's instance is referred to as the *master*, while the targetted member's is known as the *slave*. Also, since futures and future streams are instances of the same flavor, we use one set of functions and methods for both. Therefore, some operations perform slightly different tasks depending on whether a future or future stream is referenced. These differences are pointed out. Likewise, some functions and methods perform specific master or slave operations.

:future-state of *sim-io:future*

Method

Returns *t* if a reply has been received from the targetted Cast member, otherwise *nil*.

future-state *future-object* of *sim-io:future*

Function

Returns *t* if a reply has been routed back to *future*, otherwise *nil*.

:reply of *sim-io:future*

Method

Returns two values: the reply sent back to the future and the time the reply was sent. If no reply has been received, the two values returned are *nil* and *nil*. For futures, this method is non-destructive and will return the same values if used a

second time. Likewise, the *future-state* operation will continue to return *t*. For future streams this method is destructive. Each time it is used, it removes the current reply and reply-time from the stream. The *future-state* operation is then used to determine when the next reply is available.

- future-reply *future-object* of *sim-io:future*** *Function*
Returns two values: the reply sent back to the future and the time the reply was sent. If no reply has been received, the two values returned are nil and nil. For futures, this function is non-destructive and will return the same values if used a second time. Likewise, the *future-state* operation will continue to return *t*. For future streams this function is destructive. Each time it is used, it removes the current reply and reply-time from the stream. The *future-state* operation is then used to determine when the next reply is available.
- :destination of *sim-io:future*** *Method*
Returns the targetted Cast member's stagename.
- :source of *sim-io:future*** *Method*
Returns sending Cast member's stagename.
- :memo-time of *sim-io:future*** *Method*
Returns the time that this memo was sent.
- :reply-time of *sim-io:future*** *Method*
If a reply has been received, **:reply-time** returns the time it was sent, otherwise nil. For future streams this is the sending time for the current reply.
- reply-time *future-object*** *Function*
If a reply has been routed back to *future-object*, the time it was sent is returned, otherwise nil. For future streams this is the sending time for the current reply.
- receive-futures** *Function*
Returns and clears the contents of a Cast member's futurebox. If futures are present this function returns a list of futures, otherwise nil. A future is an instance of the *sim-io:future* flavor.
- :memo of *sim-io:future*** *Method*
Returns the value of the *memo* instance variable. This is the packet of information transferred from the *source* Cast member to the *destination* member.
- :send-future-reply *reply* &optional *rest* of *sim-io:future*** *Method*
For futures, *reply* is routed back to the corresponding master and *reply* is returned. Future streams can send one or more replies back to the master. A single reply or a list of replies is returned.
- send-future-reply *future-object* *reply* &optional *rest*** *Function*
For futures, *reply* is routed back to the corresponding master and *reply* is

returned. Future streams can send one or more replies back to the master. A single reply or a list of replies is returned.

Futures and future streams use *send-memo* to route the communications that occur between the master and slave components. This requires the Messenger to make an entry into its address book for each master and slave. For futures, SIMULACT automatically removes these entries when the slave sends its reply to the master. It is up to the user to *close* a future stream when its usefulness has expired in order to help reduce Messenger overhead. The following functions and method are provided for this reason.

:close of sim-io:future

Method

Sending this message to a future stream's master or slave closes the stream permanently and returns *:closed*. No further replies can be sent. All other future stream operations remain enabled.

close-future-stream *future-stream*

Function

This function closes *future-stream*, returns *:closed*, and prohibits the sending of any further replies. All other future stream operations remain enabled.

:mode of sim-io:future

Method

Returns *:closed* if the future stream has been closed, otherwise *:master* or *:slave* is returned.

future-mode *future-stream*

Function

Returns *future-streams* mode. If *future-stream* has been closed, *:closed* is returned, otherwise *:master* or *:slave* is returned.

Note that either the master or slave can close the stream at any time. It is up to the user to be aware this possibility, and to incorporate this feature in its use of future streams. For instance, Actor A may request information from Actor B by opening a future stream. Sometime later Actor A may choose to close this stream when a sufficient number of replies have been received to satisfy the request, or if the need for this information no longer exists. In either case, it is inefficient for Actor B to continue processing this request. Therefore, it makes for good programming practice for Actors to periodically test the mode of future streams in order not to waste their resources on a closed stream.

A slave may also close a stream using the same protocol as above, providing that the master periodically checks the mode in order to avoid situation of waiting for a reply from a closed stream. Note that all replies sent prior to closing the stream are retrievable in the usual manner. A slave could also notify the master that the stream has been closed by sending a reply stating this fact prior to closing the stream.

3.2 The Support Package Facility

The Support Package facility was developed to reduce memory requirements and to facilitate modularity in application code. The underlying assumption that each Cast

member has an independent environment implies that multiple copies of code are required for agents performing identical tasks. A Support Package reduces this redundancy requirement by allowing several Cast members to access the same copy of code from within their respective environments.

As in any shared memory system, an integrity violation would occur whenever a Support Package accesses or alters global information. (Global in the sense that more than one Cast member can access the same information.) To guard against these problems, SIMULACT detects the potential occurrence of integrity violations and warns the user when a Support Package tries to instantiate a global variable. Ideally, Support Packages should contain purely functional code. However, this restriction would severely constrain the code that can be placed into Support Packages.

There are two ways to use Support Packages other than for purely functional code. One way is for a Cast member to pass a local data structure as an argument to a Support Package function. If that function is "for effect", the result could then be bound appropriately. The other method requires the application programmer to use SIMULACT's *simset* functions. Basically, the *simset* functions allow Support Package code to directly alter a global variable that is present in each of the Cast member packages. (Here, global means that each Cast member's environment has its own instance of this global variable.) These functions are straightforward to use, but are built from concepts pertaining to Common Lisp's package system. Therefore, an overview of the package system will be given before introducing these functions.

3.2.1 An Overview of Common Lisp's Package System

Common Lisp's *package system* [Steele 1984] allows multiple name spaces for symbols, thus allowing for symbols with the same name to exist in different packages. A *package* is a data structure that maps symbol names to symbols. Only one package can be current at a time, and this package is bound to the variable **package**.

A *symbol* is a data object that has a separate slot (cell) for its *print name*, *value*, *definition*, *property list*, and *package*. A print name is a string used to identify the symbol. Since symbols are most commonly used to represent variables and functions in Lisp programs, they have two distinct cells for this purpose. When used as a variable, its value is stored in the value cell. Likewise, a function's definition is held in the definition cell. The property list allows the user to give symbols other attributes as well, without requiring a separate cell for each attribute. Instead, the attribute's name and value are placed into the property list. A symbol can have an unlimited number of attributes. When an attribute is accessed, the property list is searched for its name and the corresponding value returned. When a symbol is created, the package it belongs to is placed into the package cell. This value is referred to as the *home package* for this symbol.

The package system was created to allow modular programming in Lisp, without concern about conflicts between symbol names. For instance, a programmer coding in package

pkg-a could declare **x**, **y**, and **z** to be global for his own use. Meanwhile, a second programmer could declare **x** to be global in package pkg-b, without being aware of the existence of **x** in package pkg-a.

Since modular systems need to interface between modules, symbols in packages are classified as either *external* or *internal*. An external symbol is intended to be used by other packages, while internal symbols are not. Symbols are usually internal until made external using some export command.

External symbols can be accessed by other packages using its *qualified* name or through *package inheritance*. A symbol's qualified name is constructed by appending the symbol's home package name to its print name separated by a colon. For example, if **x** in package pkg-a is an external symbol, it can be accessed in package pkg-c via the qualified name pkg-a:**x**. If the symbol **x** in package pkg-a was inherited by package pkg-c, it can be accessed by just its print name. Note that a name conflict would result if package pkg-b tried to inherit **x** from pkg-a, since **x** already exists in pkg-b.

There is no way for a package to inherit the internal symbols of another package; a package can only inherit external symbols. However, internal symbols can be accessed through the use of the *double colon qualifier*. For instance, if **z** is internal to pkg-a, pkg-b can access it by pkg-a::**z**.

When one package inherits the external symbols of another, it is said to *use* that package. All packages used by a package are placed into that package's *use-list*. When a symbol is being "looked up" by a package, the package first searches through all external and internal symbols belonging to it. If the symbol is not found, the external symbols of the packages found in the use-list are searched until the symbol is found. If package pkg-a uses pkg-b, and pkg-b uses pkg-c, pkg-a does not inherit pkg-c's external symbols. This is because only pkg-b is placed into pkg-a's use-list, and not pkg-c. Also, it is possible and sometimes necessary for pkg-a to use pkg-b, and for pkg-b to use pkg-a.

3.2.2 Support Packages Restrictions

Support Packages are used to reduce memory requirements for systems where Cast members perform common tasks. They are also used to facilitate modularity in programming in exactly the same manner the package system does in Common Lisp. In fact, a Support Package is a Common Lisp package with a few additional constraints placed on it.

Global variables are not allowed in Support Packages in order to keep each Cast member's environment independent. If during the initialization of a Support Package, a defvar or some other function that instantiates a global variable is used, SIMULACT interrupts the current world initialization process and reports this problem to the user. SIMULACT also checks the validity of each Support Package each time its simulation mode changes. This ensures the detection and notification of any instantiated global variable in a

Support Package that may be incurred during the simulation.

Although global variables are not allowed in Support Packages, constants are. The programmer may use `defconstant` to instantiate a symbol that is a constant and accessible to any Cast member using that package. This is very useful method for representing static information about the system. This allows information of this sort to be specified in one place, which makes the programmer's job much simpler when this information must be changed.

`defconstant` *variable initial-value &optional documentation*

Special Form

Declares *variable* to be a constant in the Support Package. *initial-value* is evaluated and the result is bound to *variable*. It is an error if any further binding to *variable* is attempted.

documentation if used should be a string and is accessible via the `documentation` function.

Note that `defconst` cannot be used in place of `defconstant`. `defconst` allows the value of the constant to be changed without generating an error. SIMULACT will detect the use of `defconst`.

The only remaining restriction placed on Support Packages is that all symbols intended to be used by Cast members must be external. SIMULACT requires that at least one symbol in a Support Package is made external via the `export` function. If not the user is warned of this deficiency.

`export` *symbols*

Function

The *symbols* argument may be a single symbol or a list of symbols. The symbols exported by this function become external symbols of the support package. Note this is not the same `export` function found in any other package, since the `export` symbol is shadowed in all Support Packages.

When a Cast member uses a Support Package, the Support Package is placed into the Cast member's package use-list. This allows Cast member code to access these external Support Package symbols using print names or qualified names. For systems with large number of symbols, Support Package symbol "look up" can be more efficient using the symbol's qualified name.

3.2.3 The Simset Functions

SIMULACT's `simset` functions allow the programmer to write Support Package code that can directly access and alter the local environment of a Cast member. There are two requirements the programmer must keep in mind in order to use Support Packages in this manner. First, all data structures referenced by this code must exist in each Cast member's environment. Secondly, when these data structures are referenced by Support Package code, the programmer must use the `simval` and `simset` functions where `syneval` and `set` (or `setq`) would normally be used.

The principle behind the `simset` functions is based on the fact that each Cast member's environment is contained its own package. This means that when Support Package code is accessed by a Cast member, the current package is the calling member's package. The `simset` functions use the current package to access the appropriate data structures referenced in its code.

As an example, consider a simple black board application with two Actors, A and B. During Actor initialization, SIMULACT places all of A's symbols in package `actor-1` and B's symbols in `actor-2`. Both Actors use the Support Package `black-board`, which contains the black board code. This code assumes each Actor has two global variables: `*bb*` and `agent-name*`. The Support Package function `make-black-board` will be used to set `*bb*` to an instance of the black-board flavor, while `*agent-name*` is initialized directly from an initialization file.

Below is a sample of how this system could be implemented using the following four files: `BB-SUPPORT`, `BB-ACTOR`, `BB-ACTOR-1`, and `BB-ACTOR-2`. The code in `BB-SUPPORT` defines the black board system and is loaded into the `black-board` Support Package. In general, Cast members who use Support Package code have portions of their environment structured the same. In this case, file `BB-ACTOR` contains this common code and is loaded into both Actor environments. The remaining two files contain code specific to each actor, thus `BB-ACTOR-1` and `BB-ACTOR-2` are loaded into packages `actor-1` and `actor-2` respectively.

```
;;;File: BB-SUPPORT.LISP.1          Created 6/9/87 19:13:55

(defflavor black-board ((name (simval '*agent-name*))
                        (in-queue nil)
                        (out-queue nil)
                        ...
                        ))

(defun make-black-board ( )
  (simset '*bb* (make-instance 'black-board)))

(export 'make-black-board)

;;; The remaining code ...
```

```
;;; File: BB-ACTOR.LISP.1          Created 6/9/87 19:16:11

(defvar *bb*)

(defvar *agent-name*)

(make-black-board)

;;; The remaining code ...
```

```
;;; File: BB-ACTOR-1.LISP.1        Created 6/9/87 19:19:41

(setq *agent-name* "BB System A")

;;; The remaining code ...
```

```
;;; File: BB-ACTOR-2.LISP.1        Created 6/9/87 19:19:58

(setq *agent-name* "BB System B")

;;; The remaining code ...
```

If we look at the definition of `make-black-board` in `BB-SUPPORT`, we can see why `simset` is necessary.

```
(defun make-black-board ()
  (simset '*bb* (make-instance 'black-board)))
```

This function will be used by both Actors A and B to perform the following operations:

```
(setq actor-1::*bb* (make-instance 'black-board))
(setq actor-2::*bb* (make-instance 'black-board))
```

The `simset` function is able to perform these operations because each Actor resides in its own package. Furthermore, when `simset` is called the variable `*package*` is bound to the calling Actor's package. This allows `simset` to intern the `*bb*` symbol in the current package and set it to the appropriate value. Likewise, `simval` interns its argument in the current package and returns its value. This is how the `name` instance variable of the `black-board` flavor is set to the correct agent name (`*agent-name*`).

simset *variable value*

Function

When **simset** is called, the current package is the calling Cast member's package. *variable* is interned in the current package and bound to *value*. *value* is returned.

simval *variable*

Function

When **simval** is called, the current package is the calling Cast member's package. *variable* is interned in the current package and its value is returned.

simvar *variable*

Function

When **simval** is called, the current package is the calling Cast member's package. *variable* is interned in the current package and returned.

In order to implement this example without using a Support Package, each Actor package would be required to have its own definition of the black board system. This could be achieved by simply loading a file similar to BB-SUPPORT into each Actor package. This keeps the black board system modular, making updates to it straightforward.

3.3 The Peek And Poke Facility

The Peek and Poke facility can be invoked at any time when running SIMULACT as a monitoring and debugging tool. It allows the user to enter the local environment of any Cast member residing on any host throughout the network. Any part of this local environment can be examined or changed at the discretion of the user.

Peek and Poke is invoked by mousing on the *Peek & Poke* item in the Executive Director's Mode Pane. This action causes a menu displaying all host names to appear in the Command Pane (see Figure 4.a). After selecting the desired host, a second menu pops up displaying all Cast names known to that host (Figure 4.b). When the Cast member of interest is selected, the Peek and Poke window is exposed on top of the Actor and Ghost Display Area.

The Peek and Poke window is similar to the Lisp Listener, in that it executes a basic read-eval loop. Lisp forms are typed in at the keyboard and evaluated in the selected Cast member's top-level environment. The result is returned to the Peek and Poke window.

Error handling by the Peek and Poke facility is not interactive. When errors occur they are reported to the window and control is returned back to the read-eval loop. However, some edit type errors are detected prior to evaluation, which can be corrected by the user.

SIMULACT: Select Host To Peek & Poke	
BLACKFRIARS	ODEON
GLOBE	NO SELECT

a) Select host menu

SIMULACT: Select Cast Member To Peek & Poke	
CANTON	POTSDAM
OGDENSBURG	MADRID
MASSENA	WADDINGTON
LISBON	NORWOOD

b) Select Cast member menu

Figure 4. Peek And Poke Window Generation.

3.3.1 An Example Of The Peek And Poke Facility

Suppose we are developing an expert system which is designed to route local telephone calls through a small number of neighboring communities. The exact path each call takes is dependent upon the availability of resources that each community has, as well as the current distribution of phone traffic.

In our representation of the system, we model each town with one Actor. There is a knowledge base associated with each town that is used to record and retrieve information about the current status of the town and over all network. Each Actor environment binds this knowledge base to the variable *kb*.

We have just completed a preliminary model and during our first run we discover a problem. It seems the town of Madrid is not receiving its share of traffic. We suspend execution and enter Madrid's environment using Peek and Poke. We quickly determine that Madrid's knowledge base was initialized incorrectly, since it responds nil to the :available-resources query. At this point we suspect an error in its initialization file and exit SIMULACT to correct it. Figure 5 shows the actual methodology used in determining the source of the problem.

SIMULACT: PEEK AND POKE FACILITY (Cast Member: MADRID)
<Type EXIT to continue>

Command: (send *kb* :name)
MADRID

Command: (send *kb* :number-of-calls)
0

Command: (send *kb* :available-resources)
NIL

Command: exit

Figure 5. An Example Of Peek And Poke Used As A Debug Facility.

3.4 The Diary Facility

The Diary facility can be used as a debugging tool, or simply as a mechanism for post mortem analysis of system behavior. There are three levels of Diaries, all of which may be independently active. The Executive Director's Diary, when turned on, records all inter-machine messages handled. A Director's Diary records all inter-machine communications involving itself, along with the local message activity between its resident Cast members. The last type of Diary is at the Cast member level. When activated, a Cast member's Diary records its own message activity, and all screen activity initiated via the typical print and format functions.

When a Diary is activated, a corresponding file is opened in the SIM-WORLDS; directory. The default name for this file is "stagename.DIARY", where stagename is the stagename of the object for whom the Diary is being created. In all but one case, the user is asked to confirm the default file name or enter a different one. The exception being when a Diary is created during an "Old World Load".

There are three mechanisms for activating Diaries. First, during a "New World Generation" the *initialization menu* for the Executive Director, the Directors, and the Cast have an entry to turn "On" or "Off" Diary Generation. Similarly, during a "Current World Edit", each *edit menu* has this same option. The remaining mechanism for turning on Diaries is invoked by mousing on the *Turn On Diaries* option of the Mode Selection Pane.

When the *Turn On Diaries* item is selected, a tv:multiple-choose menu pops up in the Command Pane. This menu contains the stagenames of possible candidates whose Diaries

are not currently on. Each entry in this Diary menu is mouse sensitive and is highlighted when selected. Mousing on the do it option turns on the corresponding Diary for each highlighted item. Selecting the All option will turn on all Diaries just as if each item in the menu was highlighted prior to mousing on Do it. The Abort option exits this menu without turning any Diary on.

Diaries can be turned off in a similar fashion to turning them on. "Current World Edit" can be used to set the current state of Diary Generation to "Off" when editing, or the *Turn Diaries Off* choice in the Mode Selection Pane can be moused to generate a pop up menu of all Diaries. Again each item in this menu is mouse sensitive allowing the user to specify which Diaries to terminate.

3.4.1 An Example Of The Diary Facility

As an example of the Diary facility, consider the situation where Actor "ACTOR-1" resides on one host, and Actor "ACTOR-2" resides on another. Suppose at time 842 (milliseconds) ACTOR-1 sends ACTOR-2 a memo, which is the first interaction between Cast members in the emulation. The following is the Diary for ACTOR-2:

;;;File: ACTOR-2.DIARY

Created 8/23/87 19:13:55

SIMULACT's Diary Facility

:STAGENAME "ACTOR-2"
:ALIAS-NAMES NIL
:CAST-TYPE ACTOR
:SUPPORT-PACKAGES NIL
:DIARY-FLAG T
:DIARY-NAME "ACTOR-2.DIARY"
:TIME-FRAME 1000
:SCRIPT-FUNCTION BASIC-ACTOR
:INIT-FUNCTION NIL
:INT-FILES ("ACTOR")

<<Receiving memo at 1000>>

Destination: "ACTOR-2"
Source: "ACTOR-1"
Memo time: 842
Memo: "ACTOR-1 :STATUS T"

<<Retrieving memo at 1033>>

Destination: "ACTOR-2"
Source: "ACTOR-1"
Memo time: 842
Memo: "ACTOR-1 :STATUS T"

.
.
.

The first portion of this Diary contains ACTOR-2's biography, which is a description of ACTOR-2's initialization as specified by the programmer. Information such as the Actor's stagenam and script function can be very useful to help identify which agent this Diary belongs to, as well as providing some insight to its overall functionality.

Immediately following the biography is a log ACTOR-2's message activity. This log shows that at time 1000 (milliseconds), ACTOR-2 received a message from ACTOR-1. Note that ACTOR-1 sent this memo at time 842. The delay is a prevalent because SIMULACT only distributes mail between time frames. Although this memo was received at time 1000, it can be seen that ACTOR-2 did not retrieve it until time 1033. Remember, it is up to the application code to periodically check its mailbox (memos-p) for mail, and then to retrieve it when suitable (receive-memos).

This next Diary is at the Director level, and describes ACTOR-2's Director:

```
;;;File: DIRECTOR-ODEON.DIARY      Created 8/23/87 19:13:56
```

```
SIMULACT's Diary Facility
```

```
:STAGENAME "DIRECTOR-ODEON"  
:ACTORS ("ACTOR-2")  
:GHOSTS NIL  
:NUMBER-OF-ACTORS 1  
:NUMBER-OF-GHOSTS 0  
:SUPPORT-PACKAGE-P NIL  
:SUPPORT-PACKAGES NIL  
:DIRECTOR-PANE-CONSTRAINT :HORIZONTAL  
:DIARY-FLAG T  
:DIARY-NAME "DIRECTOR-ODEON.DIARY"  
:TIME-FRAME 1000
```

```
<<Received memo from Executive Director at 1000>>
```

```
Destination: "ACTOR-2"  
Source: "ACTOR-1"  
Memo time: 842  
Memo: "ACTOR-1 :STATUS T"
```

```
<<Delivering memo to Cast member at 1000>>
```

```
Destination: "ACTOR-2"  
Source: "ACTOR-1"  
Memo time: 842  
Memo: "ACTOR-1 :STATUS T"
```

```
.  
.  
.
```

Again, we see a biography, this time belonging to Director "ODEON". The log that follows is a history of message traffic handled by the Director's Messenger. Note that the Diary at this level indicates from who the message was received from, and to whom it was delivered to. It does not record when ACTOR-2 retrieves it from its mailbox.

Finally, we have the Executive Director's Diary:

;;;File: EXECUTIVE-DIRECTOR.DIARY Created 8/23/87 19:13:57

SIMULACT's Diary Facility

:STAGENAME "EXECUTIVE-DIRECTOR"
:DIRECTORS ("BLACKFRIARS-DIRECTOR" "ODEON-DIRECTOR")
:OTHER-HOST-NAMES ("ODEON")
:*TIME-FRAME* 1000
:QUANTUM 6
:DIARY-FLAG T
:DIARY-NAME "EXECUTIVE-DIRECTOR.DIARY"
:MODE-SELECT-PANE-SIZE :LARGE
:CURRENT-WORLD-NAME "DIARY-EX"
:AUTHOR "DOUG"
:AUTHOR-COMMENT "SIMPLE EXAMPLE OF THE DIARY FACILITY."
:CREATION-TIME "19:10:02, Sunday August 23,1987"

<<Received memo from Director Messenger at 1000>>

Destination: "ACTOR-2"

Source: "ACTOR-1"

Memo time: 842

Memo: "ACTOR-1 :STATUS T"

<<Delivering memo to Director Messenger at 1000>>

Destination: "ACTOR-2"

Source: "ACTOR-1"

Memo time: 842

Memo: "ACTOR-1 :STATUS T"

.
.
.

Although this is a trivial example, the power behind the Diary facility can be seen. It can be used to trace system activity and to monitor the flow of information between agents. This information can prove to be invaluable when trying to determine an agents lack of response, or wrong response to the current state of the system. Also for systems which seem to performing adequately, this information can be used to better understand system performance.

3.5 The Gauge Facility

During the development of any complex system, it is often convenient to monitor system parameters. These measurements can convey a great deal of statistical information about the system. For instance, this information can be used to characterize system performance, to monitor system resource allocation, or to measure any other system attribute. SIMULACT provides this capability to user through its Gauge facility.

3.5.1 The Gauge Pane

SIMULACT's Gauge Pane (see Figure 6) is the interface between the Gauge facility and the user. This pane is broken into two segments: one for SIMULACT's gauges and the other for user defined gauges. The gauges displayed by SIMULACT reveal the current elapsed time of the system, the current mode of operation, and the time frame ratio.

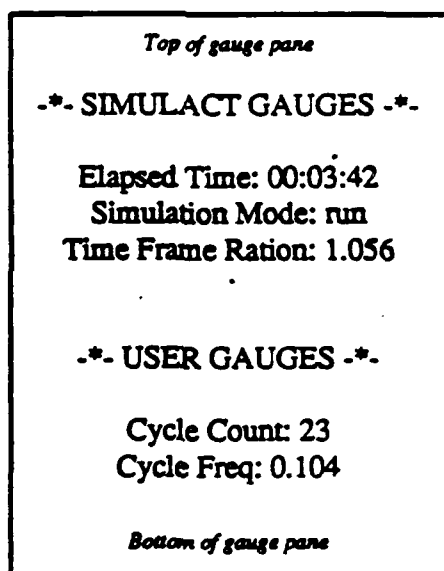


Figure 6. SIMULACT's Gauge Pane.

3.5.2 SIMULACT Gauges

Elapsed-Time

Gauge

The Elapsed-Time gauge displays the system's current elapsed time in hours:minutes:seconds. This value is bound to `simulact:*elapsed-time*` and can be accessed by the application programmer.

`simulact:*elapsed-time*`

Variable

The value of this variable is the current elapsed time of the system in

milliseconds. SIMULACT guarantees that the elapsed time associated with any given Actor is greater than or equal to this value, but less than this value plus one time frame.

Simulation-Mode

Gauge

SIMULACT's current mode of operation is displayed by this gauge. The current mode is changed via the mouse sensitive Mode Pane, which is only responsive when SIMULACT's current mode is "run" or "pause".

Time-Frame-Ratio

Gauge

This gauge is a measurement of SIMULACT's distributed performance. It is defined as:

$$\frac{(\text{elapsed wall time})}{(\text{sum of all Actor elapsed time})}$$

This ratio times the number of Actors in the system provides an estimate of how much time is required by SIMULACT to execute one time frame. For the ideal situation involving no overhead, this ratio would be 1.00, 0.50, and 0.33 for one, two, and three machine distributions respectively.

The Elapsed-Time and Time-Frame-Ratio gauges are updated automatically at the rate specified by the variable *simulact:*gauge-refresh-interval**. This interval is relative to the elapsed time of the system and defaults to 1 second. The Simulation-Mode gauge is refreshed each time SIMULACT's mode of operation is changed by the user.

simulact:*gauge-refresh-interval

Variable

The value of this variable is the interval of elapsed time that passes between Gauge Pane updates and is measured in milliseconds. Its default value is 1000 milliseconds (1 second).

3.5.3 User Gauges

Other gauges can be defined by the application programmer and placed into the Gauge Pane at any time by the *make-gauge* function. For instance, a gauge can be a permanent part of the system defined by application code, or it may be created through the Peek and Poke facility for debug purposes. In general, a gauge is used to display the value of a variable or Lisp form whose value reflects the current state of the modeled system.

make-gauge name form &rest options

Function

The *make-gauge* function instantiates and returns an instance of the *sim-io:gauge* flavor. Depending on the value of the *:local* init-option, the gauge is either global or local. If global, an entry for this gauge is placed in each Gauge Pane throughout the network, otherwise the gauge is only entered into the Gauge Pane of the Lisp machine executing this function.

name is used to identify the gauge and must be a string. The value of *name* along with the gauge's value is what appears in the Gauge Pane.

The value of a gauge is determined by *form*. There are three possible *form* types: *string*, *symeval*, or *function*. The *form* is represented as a list, whose first element is one of the keywords *:string*, *:symeval*, or *:function*.

:string

Format: (*:string*)

The *:string* type allows *name* to be entered into the Gauge Pane without an associated value. This technique can be used to enter *headings* into the Gauge Pane.

:symeval

Format: (*:symeval symbol*)

The *:symeval* type displays the value bound to *symbol* in the Gauge Pane. Note that *symbol* must be a global symbol.

:function

Format: (*:function function &rest args*)

The *:function* type passes *function* and its arguments (if any) to eval each time the gauge is evaluated.

options is a list of keywords followed by values (see section 3.5.4).

3.5.4 make-gauge Options

User defined gauges may be *global* or *local*. A global gauge is visible throughout the network in each Gauge Pane (refer to section ??). A gauge is considered local when it only appears in the Gauge Pane of the Lisp machine on which the *make-gauge* function was executed. The *:local* init-option of *make-gauge* is used to make a gauge local. SIMULACT's Elapsed Time, Simulation Mode, and Time Frame Ratio gauges are global.

:local *t-or-nil* (for *make-gauge*)

Init Option

The default is nil. If this is specified as t, the gauge will be a local gauge.

Since gauges are dynamic, their values are subject to change as the simulation advances. This means that SIMULACT must periodically re-evaluate the value of each gauge, as well as to update each Gauge Pane in order to reflect these changes. The refreshing of each Gauge Pane is done automatically by SIMULACT at a rate specified by

the value of `simulact:*gauge-refresh-interval*`. The rate at which each gauge is evaluated is set by the `:update` init-option of `make-gauge`. This value defaults to 5 seconds.

`:update interval` (for `make-gauge`)

Init Option

interval is an integer amount of time measured in sixtieths of a second used to regulate the update frequency of a gauge. If not specified, this interval defaults to 300 sixtieths of a second (5 seconds).

3.5.5 User Defined Gauge Example

As an example of the Gauge Facility, let's consider the code that produced the two User Gauges shown in Figure 6. Here we have an application that is cyclic in nature. At the conclusion of each cycle, the global variable `*cycle*` is incremented. There are two parameters of interest: the total number of cycles and the number of cycles per second. The following is a listing of code that produced these two gauges (only code specifically related to the gauges is shown):

```
;;; An Application Cyclic In Nature

;;; Global variables
(defvar *cycle* 1) ;incremented during each cycle

;;; Function definitions
(defun cycle-frequency ()
  (format nil "~5,3F" ;floating point number (x.xxx)
    (if (> simulact:*elapsed-time* 0)
      (/ *cycle* (/ simulact:*elapsed-time* 1000.0))
      0)))

;;; Make gauges
(make-gauge "Cycle" '(:symeval *cycle*))
(make-gauge "Cycle Freq: " (:function cycle-frequency))
```

3.6 The Controlled Program Wait (CPW) Facility

The controlled program wait (CPW) facility is a user invoked mechanism designed to increase system performance. The use of this facility can significantly reduce system run time, which is achieved by not processing idle Cast members during the time frame cycle. Instead, for each time frame that a member is idle, its elapsed time component is incremented by one time frame. This technique reduces the consumption of system cpu time, while preserving the elapsed time relationship among Cast members within the distributed testbed environment.

The application programmer invokes the CPW facility via the **cast-wait** function. This is typically done in the application code when it is clear that a member cannot continue until some action occurs. For example, suppose in your system there is a Knowledge Base Manager (kbm) whose job is to respond to queries. If these queries are implemented using futures, the kbm's top level function could have the following appearance:

```

      .
      .
      .
      (do () ((futures-p)))
      .
      .
      .

```

In this implementation, it can be seen that an idle kbm continues executing the **do** loop until a query is received. In an environment where a substantial amount of waiting of this nature is incurred, **SIMULACT** would consume valuable cpu time emulating these wait states.

The controlled program wait facility provides an alternative to these type of situations. For our kbm example, the following code would enhance **SIMULACT**'s performance by placing the idle kbm into CPW mode until a query is received:

```

      .
      .
      .
      (cast-wait #'futures-p)
      .
      .
      .

```

This implementation does not allow the kbm process to run again until a future is received. When it does eventually run, its elapsed time will reflect the the amount of time that it was idle.

In general, when an Actor executes the **cast-wait** function, its current time slice is immediately interrupted and the Actor's process enters CPW mode. Once in this mode, the predicate function passed by **cast-wait** is then executed at the beginning of each successive time frame. For as long as this function returns **NIL**, this process does not run and its elapsed time is incremented by one time slice for each time frame cycle.

When the predicate returns non-NIL, the process is taken out of CPW mode and resumes running where it left off.

The *cast-wiat* function can also be executed in a Ghost's environment. However, due to the differences between the roles of Ghosts and Actors, the use of the CPW facility by a Ghost is slightly different. An Actor's script function is written never to terminate and runs in its own process. This allows the CPW facility to essentially interrupt and then resume the Actor's process. On the other hand, a Ghost's script function must terminate, since it is executed at the beginning of the time frame for its affect on the current state of the emulation. Also, all Ghost script functions run in the SIMULACT process. For these reasons, when a Ghost executes a *cast-wait* there is no immediate affect. The Ghost's script function continues running until it terminates. It is at the beginning of the next time frame that the Ghost will be in CPW mode.

Once a Ghost is in CPW mode, it will be the predicate function that was passed to *cast-wait* that gets executed at the start of each successive time frame. A Ghost will remain in CPW mode until this predicate returns nonNIL. It at this point that the Ghost's script function will again be allowed to run.

cast-wait predicate &rest args

Function

Executing the *cast-wait* function places the Cast member into CPW mode. For an Actor, the Cast member's process is immediately interrupted until the Actor is no longer in CPW mode. When an Actor exits from CPW mode, its suspended process is resumed at the point of interruption. When *cast-wait* is executed by a Ghost, its script function continues running until it terminates naturally, then it is in CPW mode. When a Ghost exits CPW mode, its script function is then executed at its beginning, not at the point where the *cast-wait* was executed.

A Cast member remains in CPW mode until the predicate function applied to *args* returns non-NIL. While in CPW mode, the Cast member's elapsed time is incremented by one time frame for each time frame it was in CPW mode.

A DISTRIBUTED DEVELOPMENT ENVIRONMENT FOR DISTRIBUTED EXPERT SYSTEMS

**Douglas J. MacIntosh and Susan E. Conry
Electrical and Computer Engineering Department
Clarkson University
Potsdam, New York 13676**

Electronic Address (BITNET): DOUGMAC@CLVMS or CONRY@CLVMS

Abstract

In this paper, we describe an environment intended to aid the development of applications involving distributed problem solving. Specifically, we present a domain independent development and simulation facility which permits rapid prototyping, interactive experimentation, and ease of modification of such systems.

Our environment, called **SIMULACT**, is based on a model which regards intelligent agents as semi-autonomous problem solving agents which interact with one another by means of a message passing paradigm. This system is currently implemented on a **SYMBOLICS 3670** and on a network of **LISP** machines which incorporates both **TI EXPLORER** and **SYMBOLICS** machines.

1. Introduction

Distributed problem solving systems have received increasing attention in the AI community. Two factors have motivated this phenomenon. First, the advent of large parallel machines and the development of small, powerful microprocessor based systems have encouraged research on problems related to parallel and distributed AI systems. Secondly, research in distributed problem solving has been driven by the observation that a number of important applications are inherently distributed. Examples include distributed situation assessment, distributed sensor nets, air traffic control, and control of geographically distributed systems such as communications systems and power transmission networks.

It is easy to envision application environments in which on the order of ten to fifty semi-autonomous agents might be cooperatively solving a problem. In such an application, a distributed problem solving system would generally be implemented as a distributed system with as many independent processors as there are agents in the system. It would be prohibitively expensive to build a network of processors for the purpose of providing a testbed in which feasibility studies could be performed and initial prototype systems developed. The alternative of simulating the desired system on a single processor is not attractive, since testing a system of this magnitude on a single LISP machine would probably require time consuming simulation runs for evaluation purposes. A facility which permits a network of k machines to emulate the behavior of a network of n machines (where $n > k$) would provide an attractive alternative.

In this paper, we describe SIMULACT, a development environment for distributed problem solving systems which provides such a facility. The underlying model of problem solving which is employed regards the problem solving system as a collection of semi-autonomous agents which cooperate in problem solving through an exchange of messages. The system is modular: each agent is essentially an independent module which can easily be "plugged in" to the system. An agent's interaction with other agents in the system is totally flexible, and is user specified. Neither the form nor the content of inter-agent messages is specified by SIMULACT itself. In addition, the user can suspend execution at any time, examine the state of any agent, modify the state, the knowledge base, or even the code of an agent, and resume execution. A trace facility makes post-mortem examination of activity feasible, and a gauge facility allows the user to instrument the system in a very flexible manner.

2. Background

The architectures of the distributed AI systems that have been developed have largely been driven by the nature of the application. For example, the DVMT [Lesser and Corkill 1983] is clearly a descendent of the HEARSAY systems, and this has been natural because the signal interpretation tasks involved in vehicular tracking are similar in nature to those of speech recognition. There have been very few efforts directed towards the problem of establishing a domain independent environment suitable for the development, testing, and debugging of distributed applications.

One of the notable exceptions is MACE (Multi-Agent Computing Environment) [Gasser 1986]. This system is a development and execution environment for distributed agents. It essentially provides a tool for programming multiprocessor systems in an object oriented fashion. In MACE, agents are regarded as intelligent entities, each of which is capable of performing tasks. These agents are directed and organized by the programmer through the specification of inter-agent relationships together with high level directives and constraints on behavior. Agents in MACE are implemented as property lists of the agent name, so only one copy of an agent may reside on a given processor. By contrast, in our system multiple instances of a given agent type can be resident on a processor, thus providing greater flexibility to the user.

Among those systems that are intended for development of distributed applications, most have been designed with the intent of gaining as much speed in execution as possible. Examples of this type of system are found in Stanford's CAGE and POLIGON. Both of these systems are designed for parallel execution of applications built using them and are based on a blackboard model of problem solving. CAGE [Aiello 1986] provides a problem solving framework which is based on the assumption that development will proceed on a multiprocessor system involving up to a hundred or so processors with shared memory. The user must specify explicitly what is permitted to run in parallel. POLIGON [Rice 1986], on the other hand, is designed to be run on distributed memory multiprocessor machines involving a large number (hundreds or thousands) of processors. High bandwidth interprocessor communication is necessary for a successful implementation of POLIGON. A number of primitive operations (such as rule evaluations and blackboard updates) are automatically done in parallel.

Both CAGE and POLIGON have programming languages associated with them. As in our system, these languages facilitate the writing of application programs and provide a layer of abstraction between the user and the system's implementation details. Unlike our system, CAGE and POLIGON have been devised in an attempt to investigate issues related to exactly how much speedup can realistically be anticipated when AI programs are run in parallel environments.

In the following two sections, we discuss SIMULACT's system structure and concurrency control mechanisms. We also describe the five user interface facilities, which were designed to make SIMULACT attractive as a development environment. We conclude with a brief discussion of experiments we have performed in order to assess the degree of overhead due to SIMULACT as implemented on one and two machines.

3. System Structure

SIMULACT is a distributed system that allows n agents to be modelled on k machines, where $n > k$. Each agent runs asynchronously and coordinates its activity with that of other agents through the exchange of messages. The activities performed by each agent are assumed to be complex, so that the parallelism is coarse grained. SIMULACT allows the programmer to write code in Lisp as though there were as many Lisp machines in the network as there are agents in the distributed system being developed.

As is evident from Figure 1, SIMULACT is comprised of four component types: Actors, Ghosts, Directors, and an Executive Director. Actors are used to model agents in the distributed environment. Each Actor type is individually defined, and used as a template to create multiple instances of that Actor type. An Actor is a self contained process which runs in its own non-shared local environment. Although Actors run asynchronously, the elapsed CPU time for each actor never varies by more than one "time frame". These Actors closely resemble the entities described in Hewitt's "Actor Approach To Concurrency" [Hewitt 1986].

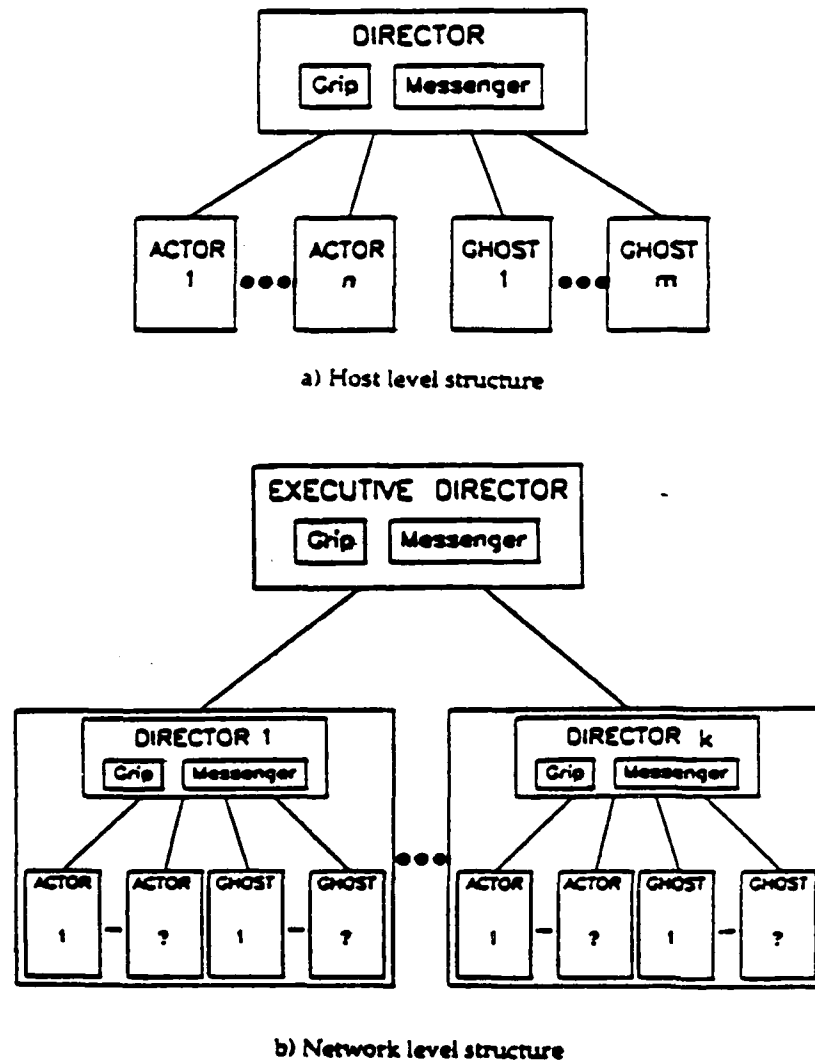


Figure 1

Ghosts are used in SIMULACT to generate and inject information into the model that would naturally occur in a "real" distributed expert system. They do not represent any physical component of the model. For example, external inputs (alarms, sensors, etc.) affecting the state of the system can be introduced via Ghosts, as well as inputs that reflect the "side effects" of the systems activities. Ghosts can also be used to

inject noise or erroneous information into the system so that issues concerning robustness can be easily investigated. The performance of an expert system can be monitored in subsequent runs through the simple modification of these Ghosts.

Due to the similarities between Actors and Ghosts, we refer to them as Cast members. Each Cast member has a unique "stagename" and a "mailbox" used by the communication facility in routing messages among members. Each also has a "script function" which defines its high level activity.

The control structure residing at each host processor in SIMULACT's distributed environment is known as the Director. The Director is responsible for controlling the activities of the Cast members at that site, and for routing messages to and from these members. These activities are assigned to the Grip and Messenger respectively. The responsibilities of the Grip range from setting up and initializing each Cast member's local environment to managing and executing the Actor and Ghost queues. The Messenger only deals with the delivery and routing of messages. When a message is sent, it is placed directly into the Messenger's "message-center". During each time frame, the Grip invokes the Messenger to distribute the messages. Whenever the destination stagename is known to the Messenger, the message is placed in the appropriate Cast member's mailbox. Otherwise, it is passed to the Executive Director's Messenger and routed to the appropriate Host.

There is one Executive Director in SIMULACT which coordinates all Cast member activities over an entire network. The Executive Director provides the link between Directors necessary for inter-machine communications, directs each Grip so that synchronization throughout the network is maintained, and handles the interface between the user and SIMULACT.

4. Concurrency Control In SIMULACT

Concurrent execution of n Actors on k machines ($n > k$) is emulated through the imposition of a "time frame" structure in execution. A time frame cycle breaks down to three fundamental parts: invocation of the Ghosts, the distribution of mail by the Messengers, and invocation of the Actors. For SIMULACT distributed over two hosts, Figure 2 depicts a representation of two time frames.

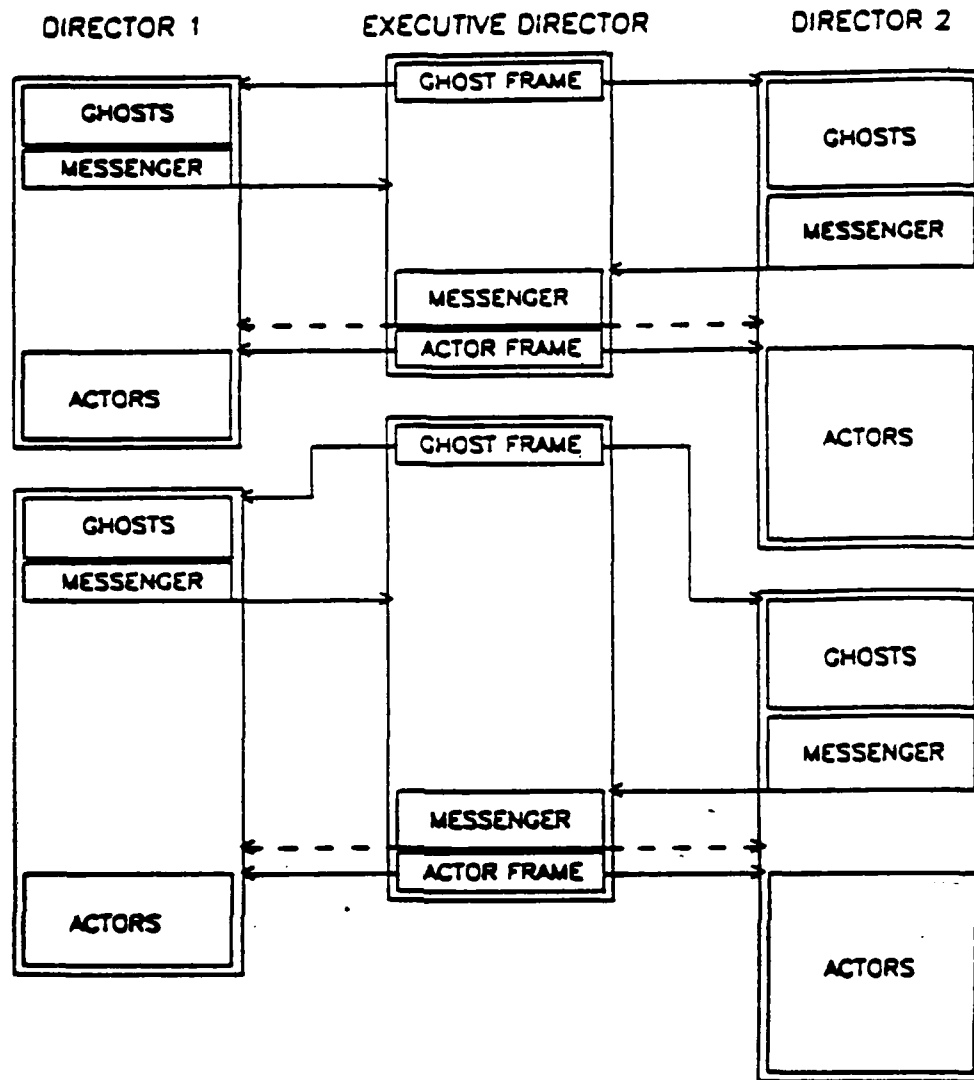


Figure 2: Activity in SIMULACT

At the start of the first time frame, the Executive Director notifies both Directors to begin executing Ghosts. (This models the occurrence of events in the world external to the distributed system.) At the conclusion of the Ghost frame, each Director automatically invokes its Messenger. The Messenger distributes all messages which were generated during the current Ghost frame, as well as all those resulting from the previous Actor frame. Mail destined for Cast members residing on the same host processor is placed in the appropriate mailboxes. The solid line extending from each Director's Messenger represents the transfer of non-local mail to the Executive Director's Messenger. In order to reduce network overhead, this transfer is done in the form of a single message. This communication always occurs, even if there are no messages to distribute, as a synchronizing mechanism for the time frame so that Actors cannot "run away". After sending this message, each Director enters a wait state until the Actor frame directive is received from the Executive Director. The dotted line directed out of the Executive Director's Messenger represents the possible distribution of inter-machine mail prior to sending the Actor frame instruction. The Executive

Director's Messenger is invoked immediately following the receipt of the last Director's Messenger communication.

Upon receiving an Actor frame command from the Executive Director, the Director's Messenger is invoked to distribute any inter-machine messages that may have been received. Next, each Actor is allowed to run for one time slice (time frame). At this point the Executive Director immediately enters its next time frame cycle, sends the Ghost frame command, and waits for all the Director Messengers to send their next synchronizing signal. Again in the second time frame of Figure 2, it is Director 2 which requires the most time to run.

5. User Interface Facilities

There are five user interface facilities that will be discussed in this section. These facilities provide mechanisms for inter-agent communication (Mail), code sharing (Support Packages), interactive monitoring and debugging (Peek and Poke), post mortem trace analysis (Diary), and runtime monitoring (Guage). These features were designed to make SIMULACT more attractive as a development environment for expert systems.

The Mail Facility

Depending on the constraints and characteristics of the expert system being developed, the application programmer constructs a network environment of intelligent agents which collectively work together towards the satisfaction of one or more goals.

SIMULACT provides several mechanisms allowing these agents to communicate without being concerned with implementation details.

In general, communication between agents occurs when one agent sends a packet of information to another, addressing the target agent by its stagename. The format of these packets is not specified by SIMULACT. Instead, it is left up to the user to formulate a syntax that is convenient in the context of the system being developed.

The "send-memo" function is the simplest mechanism one Cast member can use to communicate with another. This function accepts two arguments: the stagename of the destination Cast member and the memo to be sent. Automatically, at the beginning of the next time frame, this message will appear in the destination agent's mailbox. Each memo contains the stagename of the sending member, as well as a timetag indicating when it was sent. It is the responsibility of each Cast member to periodically check its mailbox for incoming messages.

Many communications between agents take the form of requests for information. Using the send-memo function requires that a sending agent sort its mail to retrieve the reply to a request after it has been received. Futures [Davies 1986; Halstead 1985; Rice 1986; Schoen 1986] provide an agent with a mechanism for sending a message that returns a result. A memo sent using the future facility in SIMULACT returns a data structure called a future. After the memo has been received and processed, the result

is routed back to this data structure. The sending agent uses this future to determine when the result is available, and to extract it after it has arrived. SIMULACT provides this capability through the "send-future" function.

In some cases, requests for information may not have one definitive reply. Instead, pieces of information may be returned at different times. SIMULACT allows two Cast members to establish a "future stream" between themselves for returning results over time. The user specifies criteria for determining when a future stream should be closed.

The Support Package Facility

A Support Package contains code that can be accessed by several Cast members, thus reducing memory requirements. As in any shared memory system, an integrity violation could occur whenever a Support Package accesses or alters global information. The underlying assumption concerning independent environments for each Cast member would be violated. To guard against these problems, SIMULACT detects the potential occurrence of integrity violations and warns the user when a Support Package tries to instantiate a global variable. Ideally, Support Packages should contain purely functional code. However, this restriction would severely constrain the code that can be placed into Support Packages.

There are two ways to use Support Packages other than for purely functional code. One way is for a Cast member to pass a local data structure as an argument to a Support Package function. If that function is "for effect", the result could then be bound appropriately. The other method requires the application programmer to use SIMULACT's "sim-set" functions. Basically, the sim-set function allows the Support Package to alter a global variable that is present in each of the Cast packages. The goal of the Support Package facility is to reduce overhead. Use of Support Packages does reduce the overhead, but it does so at the expense of requiring that the user have more knowledge about SIMULACT's implementation and Lisp packages [Steele 1984] than might be desirable.

The Peek And Poke Facility

This facility can be invoked at any time when running an expert system as a monitoring and debugging tool. It allows the user to enter the local environment of any Cast member and to examine or change any part of its environment. The Peek and Poke is invoked through a menu and displayed at the Executive Director's host. However, any Cast member residing on any host can be accessed.

The Diary Facility

The Diary facility can be used as a debugging tool, or simply as a mechanism for post mortem analysis of system behavior. There are three levels of Diaries, all of which may be independently active. The Executive Director's Diary, when turned on, records all inter-machine messages handled. Director Diaries record all local

communications, while Cast member Diaries record all screen activities. All Diaries are written to files to permit post mortem examination.

The Gauge Facility

This facility is used by SIMULACT to display the current elapsed run time, current mode of operation, and the "time frame ratio" which is a measure of SIMULACT's distributed performance. It can also be used as a runtime monitoring device by the user. The "make-gauge" function accepts two arguments: a string to be displayed in SIMULACT's gauge window and a function to be evaluated periodically. SIMULACT automatically evaluates this function and updates the gauge window appropriately throughout the execution of the expert system.

6. Performance Issues

The overhead incurred in managing the emulation of a distributed environment is one important measure of system performance. In this section we present preliminary results indicating the overhead incurred by SIMULACT as implemented on one and two machines. We first outline the experiments which were performed, then discuss the results obtained.

Our experiments were designed to obtain results that would assess SIMULACT's behavior as the number of messages per time frame increases. In each of the experiments, the number of messages per time frame, m , was varied over the range 0 to $10n$, where n is the number of Actors in the system. Each Actor process worked continually, consuming its total time slice allowed per time frame. Thus when $n = 0$, we measured SIMULACT's best case performance. It should be pointed out that in the distributed case where $n > 0$, the number of messages per time frame in our experiments represented entirely inter-machine communications, emulating a worst case senario.

The measurement used to represent SIMULACT's performance was a "time frame ratio" gauge. This ratio is defined as:

$$(\text{elapsed wall time})/(\text{sum of all Actor elapsed time})$$

This ratio times the number of Actors in the system provides an estimate of how much time is required by SIMULACT to execute one time frame. For the ideal situation involving no overhead, this ratio would be 1.0 and 0.5 for the one and two machine cases respectively.

Figure 3a depicts our experimental results for SIMULACT running on a single processor. Figure 3b shows the two processor results. In each case, data was collected over a range of 1 to 40 Actors per processor and a one second time frame was specified. These results are preliminary and modifications to SIMULACT have already been scheduled to enhance performance. Further testing is planned to observe the effect of three machine distribution as well as varying time frame sizes.

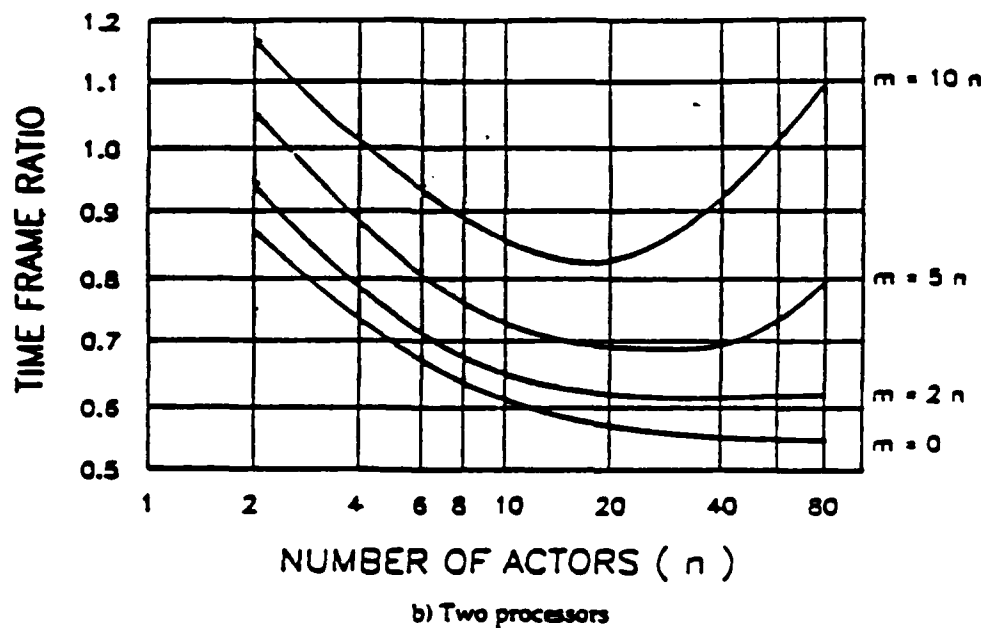
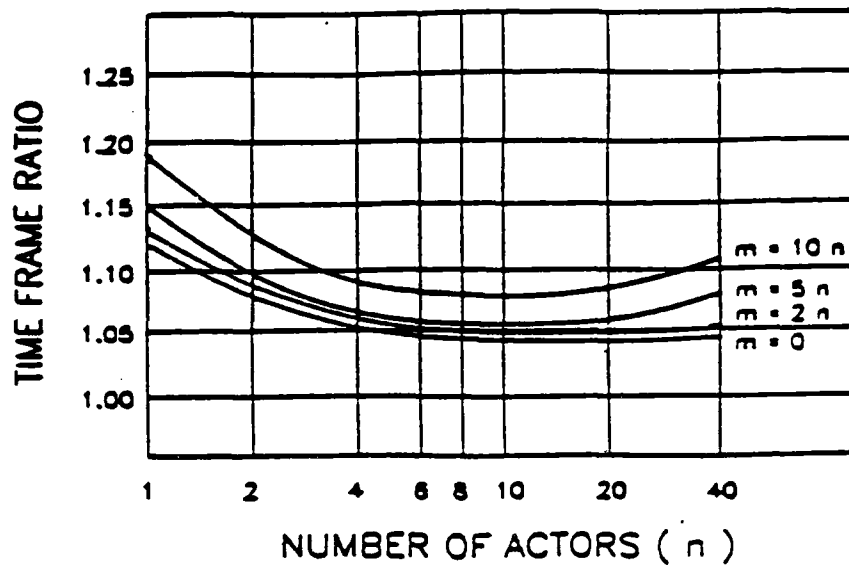


Figure 3: SIMULACT's Performance

For the single machine case with no message passing, SIMULACT's overhead approaches 4.5%. Similarly, the distributed case approaches 10% overhead. Both sets of curves indicate that as the number of messages per time frame increases, so does the overhead. In fact, between 100 and 200 messages per time frame handled by the system seems to be a saturation point for the Messenger. Currently the Messenger uses an a-list to associate stagenames with Cast members. We should see improvement when this

is implemented as a hash table lookup. Also note that the distributed case after saturation degrades at a much faster rate. One explanation for this can be deduced from Figure 2. All inter-machine messages are handled three times by different Messengers and must be sent over the Lisp machine network. Messages among agents residing at the same host processor are handled once by the Messenger and sent directly to the appropriate mailbox.

7. Concluding Remarks

In this paper, we have describe SIMULACT , an environment for the design and development of distributed expert systems. This system is currently running on a network of three Lisp machines.

SIMULACT has been particularly useful in the development of a distributed planning system [Conry, Meyer, and Lesser 1986]. It has been used to expose the nature of message traffic in this planner and to develop and debug plan generation in a distributed environment. SIMULACT is also being used as an aid in the developement of a distributed diagnosis system, an intelligent distributed knowledge base, and a distributed model for an express mail system. In each of these projects, SIMULACT 's modularity and transparency have allowed us to concentrate our efforts on the development of these agents rather than on the problems associated with managing a distributed environment.

Acknowledgments

The Authors would like to express their appreciation for the contributions of their colleagues Robert Meyer and Janice Searleman. Their suggestions and support have been invaluable. This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This Contract supports the Northeast Artificial Intelligence Consortium.

References

- Aiello, N. 1986. "User Directed Control of Parallelism; the CAGE System." In *Proceedings of the Expert Systems Workshop* (Asilomar, Pacific Grove, CA., Apr. 16-18). DARPA, 146-151.
- Brown, H., C. Tonge, and G. Foyster. 1983. "Palladio: An Exploratory Environment for Circuit Design," *IEEE Computer* (Dec.): 41-56.
- Conry, S. E., R. A. Meyer, and V. R. Lesser. 1986. "Multistage Negotiation in Distributed Planning", University of Massachusetts, Amherst Massachusetts 01003, COINS

Technical Report 86-67 (Dec.).

Davies, B. 1986. "CAREL: A Visible Distributed Lisp." In *Proceedings of the Expert Systems Workshop* (Asilomar, Pacific Grove, CA., Apr. 16-18). DARPA, 171-178.

Delagi, B. 1986. "Care User's Manual," Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Durfee, E. 1984. "A Parallel Simulation of a Distributed Problem Solving Network." M.S. Thesis, Electrical and Computer Engineering, University of Massachusetts at Amherst.

Gasser, L. 1986. "MACE, A Multi-Agent Computing Environment." USC-DPS Group, University of Southern California (Mar.).

Halstead, R.H. 1985. "Multilisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems* 7, no. 4 (Oct.): 501-538.

Hewitt, C. 1986 "Concurrency in Intelligent Systems." *AI Expert (Premier)*: 44-50.

Lesser, V.R. and D.D. Corkill. 1983. "The Distributed Vehicle Monitoring Testbed: A Tool For Investigating Distributed Problem Solving Networks." *The AI Magazine* 4, no. 3 (fall): 15-33.

Mac Intosh, D. J., S. E. Conry. 1987. "SIMULACT : A generic tool for simulating distributed systems." In *Tools for the simulation profession* (Orlando, FL., Apr. 6 - 9, 1987) The Society for Computer Simulation, San Deigo, CA., 18-23.

Misra, J. 1986. "Distributed Discrete-Event Simulation." *ACM Computing Surveys* 18, no. 1 (Mar.): 39-65.

Moon, D.A. 1986. "Object-Oriented Programming with Flavors." In *OOPSLA '86 Conference Proceedings* (Portland, OR., Sep. 29 - Oct. 2). ACM, Baltimore, MD., 1-8.

Rice, J. 1986. "Poligon, A System for Parallel Problem Solving." In *Proceedings of the Expert Systems Workshop* (Asilomar, Pacific Grove, CA., Apr. 16-18). DARPA, 152-159.

Schoen, E. 1986. "The CAOSS System." Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Report no. KSL-86-22 (Mar.).

Steele, G.L. 1984. *Common LISP*. Digital Press, Burlington, MA.

Stefik, M. and D.G. Bobrow. 1986. "Object-Oriented Programming: Themes and Variations." *The AI Magazine* 6, no. 4 (winter): 40-62.

SIMULACT: A GENERIC TOOL FOR SIMULATING DISTRIBUTED SYSTEMS

**Douglas J. MacIntosh and Susan E. Conry
Electrical and Computer Engineering Department
Clarkson University
Potsdam, New York 13676**

Abstract

This paper describes a generic tool for simulating and observing the behavior of networks of distributed agents. Our system, called SIMULACT, provides a facility for handling communication among a group of distributed agents. It does so without requiring that the application programmer be aware of SIMULACT's implementation details. The goal of the system is not primarily one of achieving speed of execution through parallelism. Instead, it is assumed that the application is inherently distributed; and a natural framework for investigating network behavior is provided.

Our system is implemented in Lisp and runs on a network of Lisp machines. The application programmer writes code in Lisp as though there were as many machines in the network as there are agents to simulate. SIMULACT is a simulator shell that provides facilities for message handling, instrumentation, and controlling the simulation. It also transparently handles all overhead associated with executing the simulation.

1. INTRODUCTION

In this paper we discuss SIMULACT, a generic tool for simulating and observing the behavior of networks of distributed systems. Unlike many other systems for simulating distributed environments [Brown et al. 1983; Delagi 1986; Schoen 1986], the goal of our system is not primarily one of achieving speed of execution through parallelism. Instead, it is assumed that the application is inherently distributed, and a natural framework for investigating network behavior is provided. Our system is useful in simulating distributed systems in which processing agents collectively work together towards satisfaction of one or more goals. It is assumed that each agent runs asynchronously and can only communicate with neighboring agents through an exchange of messages. In addition, the activities performed by each agent are assumed to be complex, so that the parallelism is coarse grained.

There are two fundamental approaches to discrete simulation: event driven and time driven. Both of these simulation strategies make use of a clock to represent the current elapsed time of a simulation and typically allow communications between simulated objects using time tagged messages. The fundamental difference between the two strategies lies in the way in which the simulation is controlled. In an event driven simulation, there usually is a data structure referred to as an "event-list". This list is used as a scheduling queue by the simulator. As the simulation advances, the scheduler removes from the event-list the entry having the earliest time tag, sets the simulation's elapsed time to this time, and initiates the evaluation of this current event by invoking the appropriate process. During the simulation new events may be generated and placed into the queue, current ones may be modified or removed, or the queue may remain unchanged. In a time driven simulation, on the other hand, the simulated elapsed time advances at a uniform rate (often termed the time frame). The clock is incremented by one time frame at each stage rather than being advanced to the time of the next scheduled process.

In order to simulate a physical system using these simulation techniques, it must be possible to model the system as a collection of discrete processes, each performing a specific task. It is then the simulator's responsibility to execute code simulating each task in an order that is consistent with that of the physical system being modeled.

There are several limitations of conventional discrete simulation systems in modeling networks of distributed agents. One is found in the time required to simulate systems having a large number of events with many interactions. It is not uncommon for a simulation of a large system to run for hours, or even days. (This phenomenon is the underlying motivation behind current research in the areas of distributed simulation architectures and techniques.) A second drawback of discrete simulation lies in the observation that there are many physical systems that cannot be adequately modeled using a discrete simulator. It may be unnatural or even impossible to view a system as a collection of discrete events which collectively perform some task. A third problem is associated with assessing the real time components associated with a simulation.

To illustrate the importance of real time issues, consider the problem of designing a distributed system that is to run on a network of cooperating microprocessors. Before such a system is built, it may be desirable to simulate its behavior over several different network configurations. It is not enough for the simulation to determine whether or not each configuration provides a viable solution to the problem. It must also provide some information as to how well each configuration performed the job (i.e., how much real time would be required by the physical system to perform the job). It is not always straightforward to determine the real time associated with a simulation. A number of discrete simulators use clocks that correspond to the event count, and not to any real time dimension [Durfee 1984; Lesser and Corkill 1983].

Each of these issues has been considered in the development of our system, and is addressed in subsequent sections. SIMULACT is written in ZetaLisp and runs on the SYMBOLICS 3600 and the TI EXPLORER Lisp machines. Our implementation makes extensive use of the flavor facility in ZetaLisp to improve data encapsulation and facilitate the modeling of environments in which a group of semiautonomous processes do not share a common memory. The current network testbed consists of one SYMBOLICS and two EXPLORERS.

2. SIMULATION STRUCTURE AND STRATEGY

SIMULACT is a distributed real time, time driven simulator capable of achieving improved run time performance through the application of event driven and parallel programming strategies. In this section, we first describe the major components of SIMULACT at a high level then discuss the simulation strategy employed by SIMULACT and some details of its implementation.

2.1 Simulation Structure

SIMULACT has a modular structure comprised of four component types: Actors, Ghosts, Directors, and an Executive Director (see Figure 1). Actors play the part of processing agents in a distributed system. Multiple instances of agents of the same type in a system are easily incorporated in SIMULACT without requiring any special code development on the part of the user. Ghosts generate information about the environment that naturally occurs in a "real" distributed system and do not represent any physical component of the distributed system being simulated. The Director is responsible for controlling the simulation and all interactions between Actors and Ghosts at each host (there is one Director for each Lisp machine in the network). Finally, the Executive Director coordinates a simulation distributed over a network of Lisp machines. In general, Actors play the role of the entities being simulated, while the Director stays backstage and tries to get realistic performance from the Actors. In the paragraphs which follow, we describe the role of each of these system components in more detail.

2.1.1 Actors

Actors in SIMULACT closely resemble the entities described in Carl Hewitt's "Actor Approach To Concurrency" [Hewitt 1986]. Hewitt's Actors are self contained entities

which work cooperatively in performing computation and can only be accessed via message passing. Sending and receiving messages are considered to be atomic operations, and messages are accepted one at a time in the order they arrive. Each message, when evaluated, may influence an Actor to create new Actors, send new communications to other Actors, or to specify the manner in which it will handle new messages.

In SIMULACT, Actors are also self contained and represent the fundamental structure used to simulate concurrency. Our Actors communicate asynchronously by routing messages through the Director. Each Actor has a "stagename" that is known by its Director and is used in routing these messages. When a Director is asked to transmit a message, it does so by either updating the appropriate Actor's "mailbox" directly, or routing the message to the appropriate Lisp machine through the Executive Director (if the destination agent resides on a different host machine in the network). When an Actor decides to read its mailbox, it has the responsibility of preserving the incoming messages and choosing which one, if any, it will currently respond to. The content and form of messages are independent of SIMULACT, and are determined by the application programmer.

The connectivity of Actors may be fixed or dynamic depending on the physical system being simulated. In a rigidly connected system, each Actor knows the stagenames of Actors with whom it can directly communicate. It is also possible for an Actor to know of the existence of a distant Actor to whom it can indirectly send a message. In simulations where the physical connectivity is allowed to change depending on the current state of the system, SIMULACT provides the capability of generating and managing appropriate stagenames on demand. This is analogous to situations in which Hewitt's Actors spawn new Actors [Hewitt 1986].

2.1.2 Ghosts

One issue which must be addressed in any simulation is the representation of events which occur in the external world and may have impact on the state of the simulation. Examples include external inputs to the simulation from its "global environment" as well as inputs which reflect the "side effects" of the simulated system's activity. Ghosts give the application programmer a facility for injecting these factors into the simulation. For example, a Ghost can use its own event-list to send an Actor a message signaling the occurrence of some event at a given time. A Ghost's event-list can easily be altered to investigate the performance of the simulated system in subsequent runs. Ghosts can also be used to inject noise, or wrong information into the simulated system so that issues associated with robustness can be easily investigated.

Actors and Ghosts are referred to as Cast members in SIMULACT, since they are very similar in structure. Each Cast member has a top level function associated with it referred to as a "script function". The script function is written by the application programmer and is invoked by SIMULACT to initiate each Cast member's simulation. Differences between these two cast types are specified (in implementation) through daemons associated with the Actor and Ghost data types. These differences arise from the fact that Actors represent the physical system and must be carefully controlled to achieve a realistic simulation. On the other hand, Ghost are considered part of the

overhead associated with the simulation.

2.1.3 Directors

SIMULACT is a distributed simulator which runs on a network of Lisp machines. Each host in the network uses a Director to control the activity of Cast members residing at that site, and to route messages to and from these Cast members. The Director assigns these activities to the Grip and Messenger respectively. The responsibilities of the Grip range from setting up and initializing each Cast member's local environment to managing and executing the Actor and Ghost queues. The Messenger only deals with the delivery and routing of messages. When a message is sent, it is placed directly into the Messenger's "message-center". During each time frame, the Grip invokes the Messenger to distribute the mail. Whenever the destination stagenam is known to the Messenger, the message is placed in the appropriate Cast member's mailbox. Otherwise, it is passed to the Executive Director's Messenger and routed to the appropriate Host.

2.1.4 Executive Director

There is one Executive Director in SIMULACT which coordinates the entire simulation over a distributed network. The Executive Director provides the link between Directors necessary for inter-machine communications, it directs each grip so that synchronization throughout the network is maintained, and it handles the interface between the user and SIMULACT.

2.2 Simulation Strategy And Implementation

In this subsection, we describe the simulation strategy employed by SIMULACT without giving extensive details regarding implementation. Mechanisms for controlling the simulation are discussed first. Since SIMULACT is implemented in Lisp using an object oriented programming style, there are two features of the system's implementation that are described in more detail. This is done in the latter half of the subsection, where we discuss message handling protocols and mechanisms for improving efficiency. The reader who is unfamiliar with Lisp and object oriented programming may find it useful to refer to [Moon 1986; Steele 1984; Stefik and Bobrow 1986].

2.2.1 Simulation Strategy

In SIMULACT, the components of the physical system being simulated are modeled solely by Actors. Only Actors need to be considered when determining the current real time associated with the simulation. For this reason, each instance of an Actor is implemented as a process on a Lisp machine. In general, an Actor's script function is written by the application programmer so that it never "terminates". At any given time, the amount of CPU time spent by the Lisp machine in executing an Actor's script function can be determined. This CPU time, along with the Actor's CPW time (Controlled Program Wait time which will be discussed in the next subsection), are used in computing the elapsed real time for a given Actor.

The Director is responsible for controlling the simulation on the host machine where it resides. Each Director has two local state variables referred to as actors and ghosts. Together these make up SIMULACT's scheduling queue local to each Lisp machine. The Director advances the simulation locally by one time frame as follows:

- (1) When the Executive Director signals the Grip to begin executing a time frame, the Director's current elapsed time is set to the minimum elapsed time of its dependent Actors.
- (2) The Grip invokes each Ghost in the ghost queue in a round robin fashion for their current effect on the simulation.
- (3) The Messenger distributes all messages present in its message center. Any message with an unknown destination is routed to the Executive Director.
- (4) The Grip signals the Executive Director that it has completed step (3) and enters a wait state. This wait state is maintained until all Grips have finished this step.
- (5) When the Executive Director signals the Grip to continue, each Actor is removed from the actor queue in a round robin fashion, and
 - if it is active, it is allowed to run for one time frame.
 - if it is not active, its CPW time is incremented by one time frame.
- (6) Go to (1).

From a network perspective the simulation can be viewed as follows:

- (1) SIMULACT's current elapsed time has value $(n)(\text{time-frame})$.
- (2) The Executive Director sends each Director a message to begin executing a time frame.
- (3) The Executive Director collects all inter-machine messages from each Director, and distributes them accordingly.
- (4) The Executive Director sends each Director a message to continue executing the current time frame.
- (5) SIMULACT's current elapsed time is incremented by one time frame ($n=n+1$).
- (6) Go to (1).

Note that for each time frame, messages are distributed after all the Ghosts have been invoked. These messages include the current ones just generated by the Ghosts, plus any

Actor messages generated during the previous time frame. It is the responsibility of each Cast member to read its mailbox in order to receive these messages, which are tagged with the time of origination. If required, the application programmer can specify a delivery delay time appropriate for the physical system being simulated.

An example of SIMULACT's simulation strategy is depicted in Figure 2. The simulation consists of 6 Actors (A, B, C, D, E, F) distributed over two hosts. During the first time frame, Host 1 allows Actors A, B, and C to run for one time frame each. Likewise, Actors D, E, and F each run for one time frame on Host 2. Neglecting all overhead (including Ghosts) one simulated time frame requires three time frame units to execute. After both machines have completed execution of the current time frame, the next time frame begins. As the length of the time frame is made shorter, the simulation model can be viewed as a collection of parallel processes, each running on an independent machine in the network. Of course, simulation using a very short time frame will increase the overhead associated with managing multiple processes running on a single machine.

Allowable time frames in SIMULACT may range from one sixtieth of a second to several seconds. The user specifies his own time frame during system initialization. For simple physical systems, SIMULACT's smallest time frame may not be long enough to expose the inherent parallelism. However, SIMULACT is designed to simulate physical systems that are complex.

2.2.2 Implementation Related Details

As has been mentioned, SIMULACT is implemented using an object oriented programming style. The flavor facility in ZetaLisp and similar linguistic constructs in languages such as CommonLOOPS and SMALLTALK [Stefik and Bobrow 1986] refer to communications among objects as being achieved through message passing. SIMULACT makes use of many types of ZetaLisp objects, not all of which are Actors or Ghosts. During the implementation of SIMULACT, we found that it was convenient to distinguish between message exchange among flavor instances and message exchange among the Cast in SIMULACT. For this reason, we say that Cast members communicate with one another by sending and posting memos. It is the application programmer's responsibility to specify the format of memos.

The difference between sending a memo and posting it is best explained using an analogy from the office environment. In an office environment, one worker would send a memo to one or more fellow workers to notify them of some specific information or request. To be sure that each worker received the memo, a copy would be placed directly in each of their mailboxes. It is up to each worker to decide what is done with the memo (i.e., read it, lose it, or throw it away). In SIMULACT, memos are sent to the mailbox associated with each Cast member. Memos are sent using a "send-memo" operation invoked by a Cast member, and received by a "receive-memos" operation invoked by a Cast member to read its mailbox.

On the other hand, if an employee had some information or a request that may be potentially valuable to some other employees, a memo could be posted at a convenient location. In this case, the worker who posts the memo does not know who may see it. Though a facility for posting memos is present in SIMULACT, it is not yet clear whether this facility has wide application.

One difficulty with the memo facility arises when a Cast member is idle waiting for an incoming memo. It would be inefficient for SIMULACT to simulate Cast members waiting. To alleviate this inefficiency, we have incorporated the use of futures and the CPW facility in SIMULACT. Futures permit Cast members to request information and continue processing until they cannot proceed without the requested data. The CPW facility has been introduced as a mechanism for increasing SIMULACT's efficiency by reducing the time spent dealing with idle agents.

Futures are used in some frameworks that support the development of multiprocessor systems, and in some parallel programming languages [Davies 1986; Halstead 1985; Rice 1986; Schoen 1986]. They provide an agent with a mechanism for sending a memo that returns a result. A memo sent using the future facility returns a data structure called a future. After the memo has been received and processed, the result is routed back to this data structure. The sending agent uses this future to determine when the result is available, and to extract it after it has arrived.

In SIMULACT, the application programmer has two future options: future-memo and future-memo-wait. The future-memo operation sends a memo to a Cast member and returns a future. The application programmer uses the functions "future-state" and "future-value" to determine when the result is available and to access it. This allows an agent to perform other tasks while waiting for a reply to a memo.

The "future-memo-wait" option is used when an agent requests information and cannot continue until it is received. This function alters the state of the Actor's process to reduce the overhead associated with implementing an Actor waiting for an incoming memo. Executing this function tells SIMULACT's scheduler to increment the Actor's CPW time by one time frame each time it is scheduled to run. Once the future has been satisfied, the Actor becomes active and will continue running beginning on the next time frame.

The controlled program wait feature gives SIMULACT the capability to speed up the simulation by not processing Actors that are waiting. At the same time, the real time component associated with each Actor is maintained. To apply this technique to situations other than memo waiting, the "cast-wait" function is provided. The programmer passes a predicate function and its arguments to cast-wait, and the CPW facility is invoked until the predicate evaluates true.

The CPW facility can also be used with Ghosts. Executing the cast-wait function within a Ghost's script function places the Ghost into CPW mode. While a Ghost is in this state, its script function is not evaluated. Instead the predicate function passed to cast-wait is evaluated. Once this function evaluates true, the ghost resumes normal operation.

3. SIMULACT'S USER INTERFACE

SIMULACT provides the application programmer with a tool for simulating and investigating the behavior of networks of agents. Application code is written for each agent in Lisp as though there were as many machines in the network as agents in the system. In this section we describe SIMULACT's user interface.

3.1 SIMULACT Initialization

SIMULACT has three initialization modes: New World Generation, Old World Load, and Current World Edit. The generation of a new world in SIMULACT is a menu driven process in which the application programmer specifies the details required to do the simulation. This includes initializing the Executive Director, the Director on each host, and every Cast member. Once the generation of a new world in SIMULACT has been completed, the user has the option of saving it to disk. The Old World Load facility automatically loads this information into SIMULACT, without involving the user. The Current World Edit option loads an old world into SIMULACT, and then allows the user to edit it. This is a convenient method for running subsequent simulations with minor alterations.

3.2 Executive Director and Director Initialization

Initializing the Executive Director configures the Lisp-machine network and sets up the user interface. The user must specify the number of hosts involved in the simulation, and set such parameters as the time frame size and the simulated start time. This allows SIMULACT to initialize the gauge facility, which is that part of the user interface which provides control over the simulation (mode select gauge), displays the current simulated time (elapsed time gauge), and allows the application programmer to monitor domain specific characteristics. The initialization of the Executive Director is not complete until all Directors are initialized.

Director initialization is done locally at each host and includes initializing the Cast and their environments. After each Cast member is initialized, the Messenger makes an entry into its "address-book". The address-book is an association list linking a Cast member's stagename to its corresponding Lisp object, and is used to route memos. Director initialization also alters the host machine's operating system to accomodate the specified time frame size.

3.3 Cast Member Initialization

Each Cast member runs in its own package [Steele 1984], so it cannot directly access any other Cast member's local state. This ensures that all communications among Cast members are directed through SIMULACT. Each Cast member has one or more initialization files associated with it. These files contain code written by the application programmer describing all activities performed by the Cast member. The Cast member's script function must be contained in this code.

Each Cast member is given a stagename during initialization. As has been mentioned, these stagenames are used by the Director's Messenger in routing memos. Each cast member is also associated with a window pane on the console's screen during initialization. The typical user interface functions (i.e., print, read, etc.) are shadowed so that the user can access each cast member easily through these windows. The collection of all Cast panes makes up the Director's window frame. SIMULACT's interface allows the user to display one Director frame at a time. It also notifies the user when a deexposed frame needs attention.

Since each Cast member maintains its own independent local state, multiple instances of the same Cast type can lead to multiple copies of code. To reduce this costly overhead SIMULACT allows Cast members to use Support packages.

3.4 Support Packages

A Support package contains code that can be accessed by several Cast members, thus reducing memory requirements. As in any shared memory system, a problem could arise whenever a Support package accesses or alters global information. The underlying assumption concerning independent environments for each Cast member would be violated. To guard against these problems, SIMULACT automatically detects their potential occurrence and warns the user when a Support package tries to instantiate a global variable. Ideally, Support packages should contain purely functional code. However, this restriction would severely restrict the amount of code that can be placed into Support packages.

There are two ways to use Support packages other than for purely functional code. One way is for a Cast member to pass a local data structure as an argument to a Support package function. If that function is "for effect", the result could then be bound appropriately. The other method requires the application programmer to use SIMULACT's "sim-set" function. Basically, the sim-set function allows the Support package to alter a global variable that is present in each of the Cast packages. The goal of the Support package facility is to reduce simulation overhead. Use of support packages does reduce the overhead, but it does so at the expense of requiring that the user have more knowledge about SIMULACT's implementation than is desirable.

4. CONCLUDING REMARKS

In this paper, we have described SIMULACT, a generic tool for simulating and observing the behavior of networks of distributed processing agents. This system is currently running as one component in a testbed for investigating problems in distributed artificial intelligence. Our current network configuration contains three Lisp machines.

The system has been particularly useful as a tool in the development of a distributed planning system [Conry, Meyer, and Lesser 1986]. It has been used to expose the nature of message traffic in this planner and to develop and debug plan generation in a distributed environment. SIMULACT is also being used as an aid in the development

of a distributed diagnosis system. We have found that its modularity and transparency permit us to concentrate on the development of agents which exhibit the desired characteristics rather than on the problems associated with managing the distributed environment.

Acknowledgments

The Authors would like to express their appreciation for the contributions of their colleagues Robert Meyer and Janice Searleman. Their suggestions and support have been invaluable. This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This Contract supports the Northeast Artificial Intelligence Consortium.

REFERENCES

Brown, H., C. Tonge, and G. Foyster. 1983. "Palladio: An Exploratory Environment for Circuit Design," *IEEE Computer* (Dec.): 41-56.

Conry, S. E., R. A. Meyer, and V. R. Lesser. 1986. "Multistage Negotiation in Distributed Planning", University of Massachusetts, Amherst Massachusetts 01003, COINS Technical Report 86-67 (Dec.).

Davies, B. 1986. "CAREL: A Visible Distributed Lisp." In *Proceedings of the Expert Systems Workshop* (Asilomar, Pacific Grove, CA., Apr. 16-18). DARPA, 171-178.

Delagi, B. 1986. "Care User's Manual," Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Durfee, E. 1984. "A Parallel Simulation of a Distributed Problem Solving Network." M.S Thesis, Electrical and Computer Engineering, University of Massachusetts at Amherst.

Halstead, R.H. 1985. "Multilisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems* 7, no. 4 (Oct.): 501-538.

Hewitt, C. 1986 "Concurrency in Intelligent Systems." *AI Expert (Premier)*: 44-50.

Lesser, V.R. and D.D Corkill. 1983. "THE DISTRIBUTED VEHICLE MONITORING TESTBED: A Tool For Investigating Distributed Problem Solving Networks." *The AI Magazine* 4, no. 3 (fall): 15-33.

Misra, J. 1986. "Distributed Discrete-Event Simulation." *ACM Computing Surveys* 18, no. 1 (Mar.): 39-65.

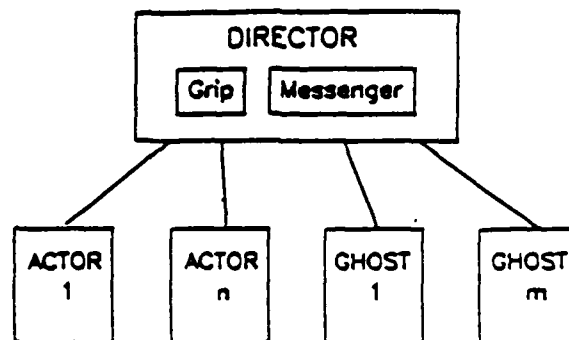
Moon, D.A. 1986. "Object-Oriented Programming with Flavors." In OOPSLA '86 Conference Proceedings (Portland, OR., Sep. 29 - Oct. 2). ACM, Baltimore, MD., 1-8.

Rice, J. 1986. "Poligon, A System for Parallel Problem Solving." In Proceedings of the Expert Systems Workshop (Asilomar, Pacific Grove, CA., Apr. 16-18). DARPA, 152-159.

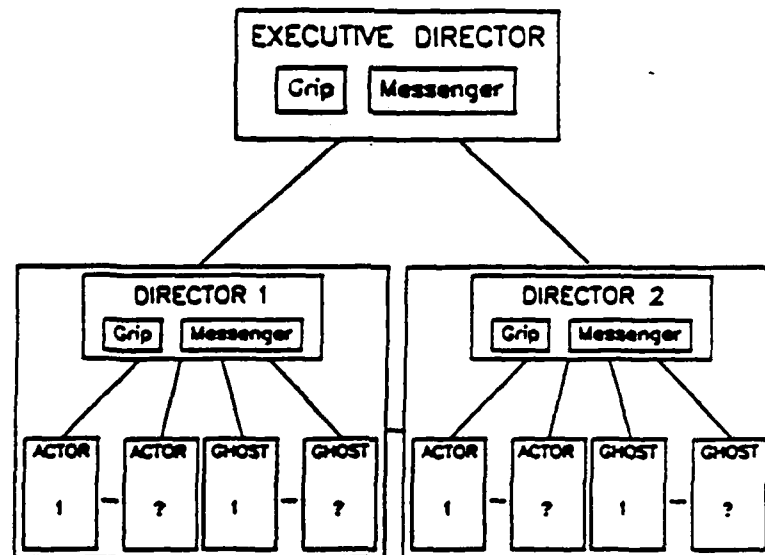
Schoen, E. 1986. "The CAOSS System." Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Report no. KSL-86-22 (Mar.).

Steele, G.L. 1984. Common LISP. Digital Press, Burlington, MA.

Stefik, M. and D.G. Bobrow. 1986. "Object-Oriented Programming: Themes and Variations." The AI Magazine 6, no. 4 (winter): 40-62.



a) Host Level Structure



b) Network Level Structure

Figure 1 SIMULACT's Simulation Structure

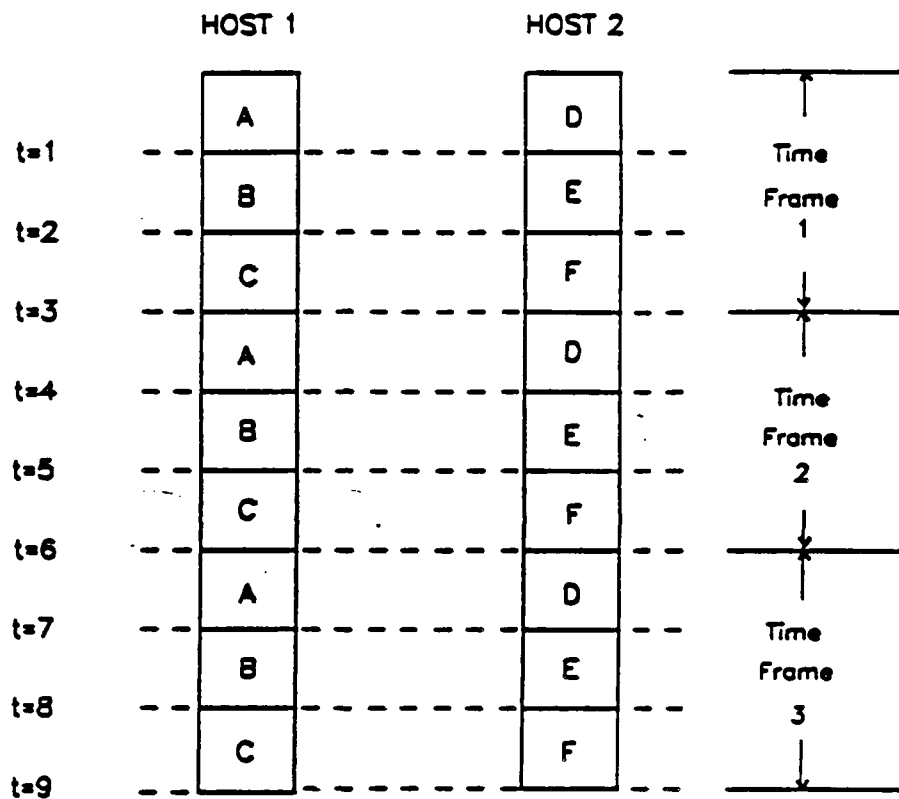


Figure 2

THE ROLE OF KNOWLEDGE-BASED SYSTEMS IN COMMUNICATIONS SYSTEM CONTROL

by

Robert A. Meyer
Electrical and Computer Engineering
Clarkson University
Potsdam, New York 13676

and

Charles Meyer
Rome Air Development Center, DCLD
Griffiss Air Force Base
New York 13441-5700

Abstract

The role of knowledge-based systems in communications system control is presented in this paper as a collection of distributed, multiple agent systems serving as intelligent advisors to human controllers. Based on an analysis of system control functions, and interviews with field personnel, we describe a model for communications system control in the Defense Communications System. Using this model we have designed an architecture for a diversely distributed, multiple agent, knowledge-based system. A simulation testbed has been implemented to support experimental development and testing of the components of this architecture in a network of Lisp machines.

Introduction

The purpose of this paper is to describe one role which knowledge-based systems may play in the future control of the Defense Communications System (DCS). The work described here is based on the completion of the first phase of a five year research program designed to answer fundamental questions about the use of knowledge-based systems in communications network management and control. We have developed an architecture for a diversely distributed, multi-agent system in which each component is a specialized and localized knowledge-based system designed to provide assistance to the human operator and to cooperate with similar such systems performing other functions and/or located in physically separate facilities. This view of the role of a knowledge-based system as a collection of autonomous, cooperating independent specialists is an important characteristic of our approach to network management and system control for the DCS in the future.

Modern communications systems, such as the DCS, are highly complex collections of equipment and transmission media which currently require, in varying degrees, human intervention for control. The control task is one which requires extensive, specialized knowledge and the ability to reason using this knowledge in solving problems. In the past, system control has been a difficult area to automate because the number of situations which may arise and alternative solutions available are very large, and thus traditional, purely algorithmic approaches have been found lacking.

In this paper we first describe a model for communications system control, based on the DCS in Europe, and identify specific control tasks needed in this environment. We find three fundamental kinds of knowledge-based problem solving activities are required: (1) data interpretation and situation assessment; (2) diagnosis and fault isolation; and (3) planning to find and allocate scarce resources for restoral of service in the event of an outage. In addition to this functional distribution of problem solving activities, our model requires a spatial distribution of control as well. We present an architecture designed to meet these requirements which consists of a distributed knowledge-based system built on a community of problem solving agents. Each agent is a functionally specialized knowledge-based problem solver at a specific site. These agents coordinate and cooperate to solve global problems among themselves, crossing functional or spatial boundaries as required.

The reason for studying distributed problem solving lies in the observation that humans often rely on teams of people to solve complex problems. Within a team there is usually a division of labor so that each member of the team is a specialist on some part of the problem. Each of these specialists has only a limited perspective about the overall problem, and each finds that he can only deal with those aspects of the problem for which he is responsible through cooperation with others on the team.

Distributed artificial intelligence is concerned with problems that arise when a group of loosely coupled problem solving agents works together to solve a problem [1]. These agents have characteristics that closely parallel those mentioned above:

functional specialization, local perspective, and incomplete knowledge. Each agent uses its own local perspective in performing its tasks, though it may have a need for some knowledge about another agent's local perspective.

At the present time a testbed has been implemented which supports simulation of multiple agents on one or more physically distinct Lisp processors. Detailed design and implementation of specific agents is in progress. We believe the paper has two chief contributions to make at this time. First, based on in-depth problem domain analysis, including field site visits and interviews with operating personnel, we have identified specific problem solving tasks which we believe are suitable for a knowledge-based system. Furthermore, our design for this system incorporates new ideas in distributed problem solving: specifically, a diversely distributed problem solving architecture which supports coordination and cooperation among functionally and spatially distinct agents.

The Communications System Control Model

In our investigation of knowledge-based systems for communications system control, we have concentrated on the European Theater of the DCS. The DCS is a large, complex communications system consisting of many component subsystems. It provides the long-haul, point-to-point, and switched network communications for the Department of Defense. We have chosen the European Theater for several reasons. The DCS network structure in Europe is particularly interesting for the study of distributed problem solve paradigms. It consists of a large number of sites (about 200) which are interconnected in an irregular structure. It is currently controlled by close cooperation and coordination among a group of highly skilled human controllers distributed throughout the system. The wide variety of transmission media and communications equipment in use has given rise to the need for sophisticated problem solving tools to assist these human operators in providing the best possible control of the system.

A careful analysis of the DCS reveals that it must be viewed from a multidimensional perspective. The overall organization of the DCS is layered, consisting of three basic layers: transmission facilities, circuits and networks. Each of these layers may be further subdivided into component subsystems. Transmission facilities may be either terrestrial or satellite. Terrestrial transmission is based on either analog or digital channels, multiplexed into groups or digroups, and then into supergroups which are transmitted over communications links from one station to another. The most common transmission medium used is line-of-sight (LOS) microwave; however, there are also tropo-scatter, fiber optic, and cable links used. Satellite transmission facilities are also used, primarily for transoceanic links.

The transmission facilities form the backbone structure over which the second layer, consisting of circuits and trunks, is built. These are predominately dedicated circuits between users. Circuits may traverse several stations following fixed paths. There are a number of key data items which are associated with each individual circuit or trunk

and which are important in the performance of system control. These include the user priority level, the restoration priority, and the quality of service required.

Networks form a third layer of the DCS organization. There are three general categories of networks: circuit switched voice, packet switched data, and dedicated or special purpose networks. These networks rely on trunks provided by the transmission subsystems to interconnect the switches.

Another perspective of the DCS is equipment oriented. The DCS consists of a very large inventory of communications equipment, such as modems, multiplexers, radios, switches, etc. Each equipment item has certain distinguishing characteristics including its function within the overall system, its status signals (which may be monitored and made available to system controllers), and its control capabilities (which provide the mechanism for implementing desired control actions on the system). Knowledge about equipment is vital to problem solving agents attempting to control the system, and cuts across the layered organization described above. For example, a particular multiplexer may be a part of a transmission facility, as well as a part of one or more circuits, and a part of one or more networks.

The final dimension along which the DCS may be analyzed is its organization for monitoring and control. Data is collected in real-time or near real-time at each site about the operating state of the equipment at that site and about the quality of signals being transmitted to the site. In a completely centralized control system, this data would be transmitted to a single control center for interpretation by the system controllers. However, our model of the DCS is based on the concept of a distributed control system, in which certain sites are identified as either sub-region, region, or area control centers. Each control center has primary responsibility for a subset of the other sites within some geographic neighborhood. Each control site receives data from the sites within its region of responsibility.

Why are we interested in a distributed control system? The primary reason is that such systems are inherently more robust and survivable in the presence of equipment failures or externally applied threats. Total reliance on a centralized control site would mean a total loss of system control in the event of a complete failure at or loss of the central site, or in the event of isolation of that site from the rest of the system. Partitioning the system into regions is also an advantage in that the control problem may be made modular, and thus easier to solve. Local problems (ones which arise internally to a region and have no impact outside the region) may be solved within the local region, and do not become visible to the other regions.

In accordance with the future plans [2] of the Defense Communications Agency (DCA), the DCS system control is to be organized in a five level hierarchical structure. Beginning at the lowest level and moving up, each level in this hierarchy represents a broader view of the DCS, a larger geographic area, a greater responsibility and a higher authority. Level 5 (the lowest level) represents stations or facilities at which either a technical control or patch and test capability exists, or an access switch exists, or an earth terminal for a satellite link exists. Level 4 represents either a major

technical control facility or nodal switch. Level 3 represents a subregion control center (SRCF). Level 2 corresponds to theater level control and may be either an area communications operations center (ACOC) or alternate ACOC. Level 1 is the worldwide Defense Communications Agency Operations Center (DCAOC). For the purposes of this discussion, we are concerned with level 3 and lower levels. These are the levels most closely associated with the real time or near real time control of the system. Within the European theater approximately 13 SRCFs are expected to be established. Thus, it is at this level (level 3), or lower, that the need for cooperative problem solving is likely to be the greatest.

System Control Tasks

System control is defined [3] as the process "... which ensures user to user service is maintained under changing traffic conditions, user requirements, natural or manmade stresses, disturbances, and equipment disruptions on a near term basis." System control incorporates five major functions: facility surveillance, traffic surveillance, network control, traffic control, and technical control. Each of these functions will be described in more detail and related to specific problem solving activities.

Three distinct problem solving activities have been identified within the five major functions of system control. We refer to these activities as performance assessment (PA), fault isolation (FI), and service restoral (SR). A general task description for each of these is given below and related back to one or more of the five functions of system control.

Performance Assessment (PA)

Performance assessment may be viewed as a problem in data interpretation and situation assessment. Since data is available only on a distributed basis, coordination must take place among the PA agents in order to arrive at a coherent view of the state of the communications system. The facility surveillance and traffic surveillance functions of system control are included within the PA activity. Real time equipment, transmission network, and traffic data are measured and collected to provide the controller with the information needed to determine the status of the transmission system and facilities, the quality of communications circuits and network performance. Trouble reports from users are also significant inputs to this activity.

The goal of PA is to formulate a local view of system status and performance, and to identify as quickly as possible the impact of any observed deviations from normal operating conditions. The PA agent is responsible for determining the need to invoke either fault isolation and/or service restoral agents. Since few problems are likely to be localized within the area of responsibility of a single SRCF, the PA agent must also communicate with similar agents in neighboring areas to arrive at a consistent assessment of status throughout the system.

Fault Isolation (FI)

The fault isolation task is a diagnostic activity. It is concerned with identifying the specific cause and location of faults within the communications system. The term "fault" is used in a very broad sense to mean either a complete outage of service or a degradation in quality or performance. The FI agent responds to reports of known or suspected faults determined by PA. Much of the same data available to PA is also used in the FI activity, however the analysis is carried out in greater depth, and the cause-effect relationship is emphasized. In some instances the immediate results of FI may be inconclusive and will require additional testing to resolve ambiguities in the data. The FI agent should be able to direct the initiation of automatic test functions where such functions are available. Knowledge about specific equipment types, the meaning of test results and the current subsystem state are all used by FI to infer the probable cause of the failure.

The FI agent incorporates the in-depth analysis aspects of facility and traffic surveillance as well as the testing aspects of technical control. Coordination and cooperation with similar agents at intermediate or distant end stations involved in a faulty link, trunk, circuit, or network are often necessary to determine the cause and location of a fault.

Service Restoral (SR)

Service restoral is a plan generation and selection activity which recommends a set of specific control actions needed to restore user service. These actions may involve alternate routing of trunks or circuits, switch control (such as code cancellation, code blocking, modification of routing tables, etc.), or transmission system configuration control (such as reallocation of equipment, use of backup or spare equipments, etc.). The network control, traffic control, and some aspects of the technical control functions are encompassed in the SR activity.

System Architecture

The system architecture which we describe has been designed to support multiple problem solving agents distributed both functionally and spatially. This discussion addresses three primary facets of the architecture: (1) the role of this distributed knowledge-based system in the context of human network controllers in the DCS; (2) the structure of the intelligent system residing at each node; and (3) the overall structure of the distributed problem solving system as it is incorporated in the simulation testbed.

Any knowledge-based distributed problem solving system actually incorporated in the DCS field environment would be most accurately viewed as an intelligent assistant to the system controllers. It would perform the tasks of filtering the data received, and analyzing and interpreting it as well. Based on these interpretations, it would suggest alternative restoral plans and advise the humans as to which equipments are probably

malfunctioning and what control actions might be appropriate. The human operator would have the responsibility of directing service restoral, dispatching service personnel, and initiating control actions. In addition, the human retains the privilege of preempting the system at any time. As an intelligent assistant, the system would relieve the controller of many tedious tasks. It would also provide a vehicle for training activity that can be utilized by new personnel.

As the discussion in the previous section indicates, several modes of problem solving must be performed at each site corresponding to a control facility of the communications system. Performance assessment is an ongoing data interpretation and situation assessment task. Fault isolation is a diagnostic activity and is initiated when PA has determined the possible existence of an equipment failure. Service restoral is a plan generation task, and is also initiated by PA. These tasks are relatively independent and may be active concurrently; however, they must share the same base of local knowledge concerning the status of the communication system and its expected behavior. An implicit assumption is that each control site only maintains detailed knowledge about the communications equipment and circuits which are within its region of responsibility. Our node architecture represents a decomposition of nodal problem solving activity into these three primary tasks and assumes the local knowledge base contains only very limited knowledge about the system outside the local area.

The structure of this architecture for distributed problem solving reflects distribution in two dimensions. At a local level, the system is seen as a number of functionally specialized agents that cooperate in a loosely coupled fashion. These agents comprise a local participant in a network-wide team of problem solvers. At the global level, the system may be viewed as a group of relatively independent, spatially distributed problem solving systems cooperating to solve a collection of problems. One of the systems is composed of the group of fault isolation agents. The fault isolation agent at each node cooperates with its counterparts at other nodes in solving the fault isolation problem for the communications system. In a similar fashion, the service restoral and performance assessment agents may be regarded as distributed problem solving systems in their own right.

An important feature of this architecture is the concept of a local, shared knowledge base. Although each problem solving agent has its own, private knowledge about how to perform the specialized problem solving activity for which it is responsible, much of the knowledge needed for problem solving is related to the communication system. This knowledge describes the network structure and organization, the details of different equipment types, and what is known about the current state of the communications system. Since this knowledge is spatially distributed, and shared among the other functional agents at the same local site, it is implemented as a single local knowledge base. The details of knowledge representation and structure of this knowledge base have been presented elsewhere, [4] and will not be repeated here. Suffice it to observe that since this knowledge base contains dynamic information, and is shared by multiple, independent agents, some sort of access control is necessary. This is provided in the form of the knowledge base manager.

The knowledge base manager serves as an intelligent interface agent between the knowledge base and the three primary agents, PA, FI, and SR, which are operating concurrently. The knowledge base manager embodies knowledge about the local knowledge base itself. For example, it knows the limits of that knowledge base in the sense that if a query for information is made which requires knowledge not locally available, the knowledge base manager issues a message requesting the information from an appropriate distant node(s). The knowledge base manager is also knowledgeable about the time sensitivity of some parts of the local knowledge base and is thus able to make intelligent (as opposed to arbitrary) decisions in controlling concurrent requests for access. Finally, the knowledge base manager has the responsibility for maintaining consistency in the local knowledge base. In order to perform these functional tasks, the knowledge base manager may be viewed as an agent having meta-level knowledge.

The local architecture thus consists of four knowledge-based components, cooperating on a local basis to maintain an informed state of awareness about the local operating conditions. This is shown in Figure 1 below.

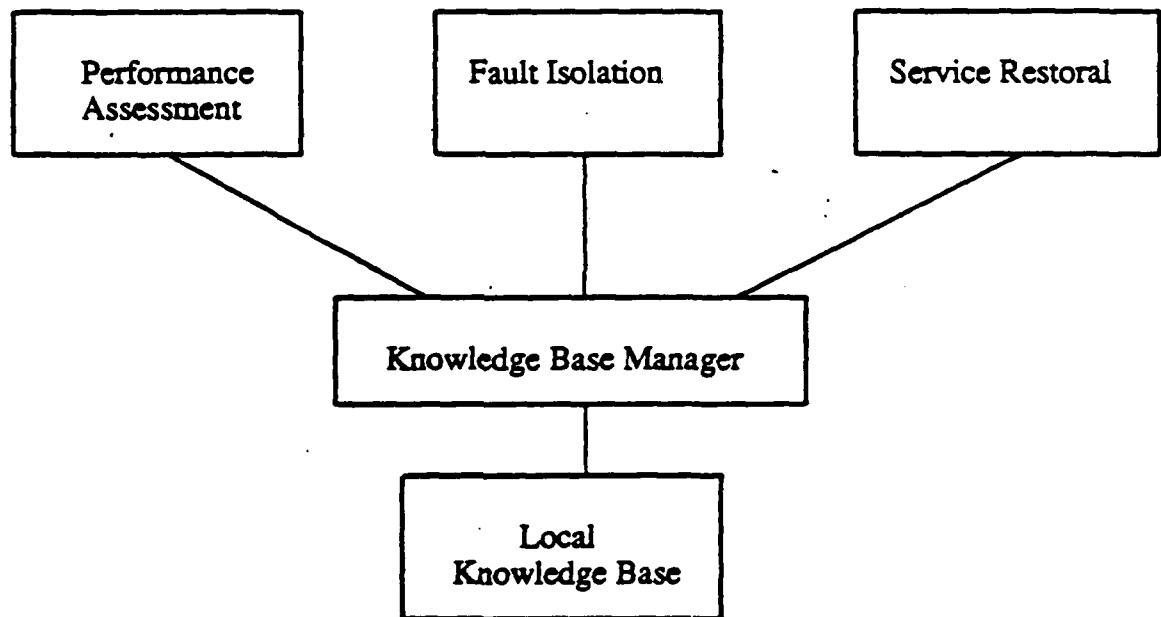


Figure 1. Control node architecture

Interagent communication is supported in the form of simple messages called memos. A memo may be sent to one or several recipients and may be a request for information or an action to be taken, a response to a previous request, or some unsolicited information shared by an agent with others. Additional components not shown in above diagram are those which interface with the external world, such as interaction with the operator, and real-time data collection.

Concluding Remarks

A generic simulation testbed, SIMULACT, has been implemented [5] to support experimental testing and evaluation of alternative design approaches for the distributed, multiple agent knowledge-based system described here. This testbed will become a tool for investigating a number of significant issues in the design of cooperative, distributed problem solvers. The goal of this research is to develop a better understanding of the fundamental problems which arise in building a distributed problem solving system. We believe the problem solving environment demanded by the critical need for defense communications under a wide variety of stressful conditions is most likely to be satisfied by a distributed, knowledge-based system, and thus the results of our research should provide guidance in meeting this need in the future.

Acknowledgments

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

References

1. Davis, R. and R.G. Smith, "Negotiation as a Metaphor for Distributed Problem Solving," Artificial Intelligence, Vol. 20, pp. 63-109. Jan 1983.
2. "Defense Communications System Control Plan (Europe)," Defense Communications Agency. Sept 1984 (draft).
3. "DCA Circular 310-70-1", Vol I, II, and III, Defense Communications Agency.
4. Conry, S.E., R.A. Meyer and J.E. Searleman, "A Shared Knowledge Base for Independent Problem Solving Agents," Proc. of the Expert Systems in Government Symp., McLean, VA. Oct 1985.
5. MacIntosh, D.J. and S.E. Conry, "SIMULACT: A Generic Tool for Simulating Distributed Systems," Eastern Simulation Conference, Orlando, FL. April 1987.

MACHINE INTELLIGENCE FOR DoD COMMUNICATIONS SYSTEM CONTROL

by

Gerald Michael Adams
and
Charles N. Meyer
DCLD/ Rome Air Development Center
Griffiss Air Force Base, NY 13441

Robert A. Meyer
Electrical and Computer Engineering Department
Clarkson University
Potsdam, NY 13676

Abstract

Modern military communication systems are complex, multi-layered networks which demand an integrated and automated approach to network management and system control. Effective system control requires interpretation of operating data to assess network status, planning to reconfigure network resources in the event of outages, and decision making to effect appropriate control actions. To meet these needs, the U.S. Air Force, in conjunction with the Defense Communications Agency (DCA), is investigating the use of machine intelligence techniques to provide real-time expert assistance to the human operators responsible for the management and control of the Defense Communications System. This paper describes an evolutionary approach to the development of these intelligent assistants, in which modular hardware and software systems are being developed and tested in an incremental progression of closely coupled projects.

1. Introduction

1.1 The Defense Communications System

The Defense Communications System (DCS) is the primary long-haul communications system for the U.S. military forces. It embodies both Government owned and leased facilities within the U.S. and abroad. These facilities encompass a wide range of networks, including direct dedicated lines, and circuit, message and packet switched networks. The DCS provides both voice and data communications services at multiple levels of security. As a result of continuing efforts to improve the quality and reliability of service, the DCS consists of a large variety of equipment types, spanning technologies from basic analog systems, such as the low data rate voice channel teletypes, to modern digital statistical multiplexers.

The DCS consists of stations interconnected by links. Each *station* has of a collection of communications equipment, typically associated with a group of local users, for example at an airbase. Of course some stations have no local users, but serve as repeaters or relay stations, or as switching centers. Each *link* represents the transmission medium interconnecting two stations, such as a microwave radio channel, a satellite channel or cable. This station/link model of the DCS represents only the first "layer" - the *transmission subsystem* - in a multi-layer structure. The transmission layer is the base upon which other networks are constructed. For example, AUTOVON is the worldwide circuit switched voice network for the military. The long-haul interswitch trunks needed for AUTOVON are provided by the transmission subsystem.

The management and control of this complex communications system is clearly an important task. We rely heavily on fast, secure, high quality communications both during peace time and during conflict. We have come to expect near 100% availability of communications resources. The job of providing the management and control of these resources to meet these expectations is called *Communications System Control*.

1.2 Communications System Control

The objective of system control is to maintain end-to-end user communications connectivity under a variety of adverse conditions. These range from routine minor equipment malfunctions to major destruction of facilities due to hostile actions. Effective communications system control requires managing network and transmission resources so as to make the best utilization of available assets given the priorities of the current context. It should be emphasized that this is a very dynamic environment since actions external to the communications system may bring about significant changes in traffic loading and priority of users.

Currently, the structure of this control system consists of two complementary parts. The transmission subsystem is predominately controlled in a decentralized manner from *tech control facilities* (TCF) distributed throughout the network. Each TCF is staffed on a 24 hour basis by highly trained military personnel known as "tech

controllers". Generally TCFs are located outside the United States, since within the U.S. the transmission subsystem is leased from common carriers, and is not under direct government control. Networks, such as AUTOVON or DDN (the Defense Data Network), are controlled from centralized control centers. These are either the Area Communications Operations Center (ACOC) in the European and Pacific theaters, or the DCA Operations Center (DCAOC) in the U.S. There is a strict hierarchical relationship among these control system elements; the DCAOC is at the top, having a worldwide view, the ACOCs each have a theater-wide view, and each TCF has a local perspective.

There are important changes being made to this current structure. The objective control structure for the future DCS incorporates the concept of a subregion and a subregion control facility (SRCF) [1]. The SRCF is intended to distribute the control capability of the various networks thus providing a more survivable control system. The SRCF is also expected to collect transmission subsystem monitoring data remoted from several TCFs and assume some degree of transmission control. The SRCF thus provides a mix of both distributed and centralized control and management capabilities. The SRCF will also provide the means for integrating transmission monitoring and control and network management functions.

1.3 Functional Requirements

In this section we examine, at a high level, the functions required to achieve the objectives of communications system control. We will use the term "*system controller*" in a generic sense to mean any combination of one or more humans and machines collectively charged with the responsibility for achieving the system control objectives.

First, the system controller must form an assessment of the current global network status. This involves interpretation of data collected from throughout the network and correlation of information from disjoint or dissimilar sources. For example, a link failure in one part of the network might have a major impact on the traffic loading of a switch in another part of the network. Under normal conditions this additional loading might not be critical. However, if other conditions (perhaps completely external to the communications system) were also contributing to an increased traffic load at this same switch, then a control action might be necessary to prevent an overload situation. Recognition of this relatively simple situation requires interpreting both transmission alarm data and switch traffic data, assessing the impact of the link failure, and correlating this conclusion with the assessment of the current switch condition.

A second functional requirement for the system controller is to formulate plans for alternative actions which would alleviate problem situations of the type described above. Planning is the primary function which attempts to make the best utilization of available communications assets.

A third requirement is for a diagnostic capability. While assessment recognizes changes in network status, and planning provides alternatives to alleviate problems, a longer term solution must be sought which attempts to diagnose the cause of the problem,

so that it may be corrected. In many cases diagnosis is simply a matter of tracing circuit paths until the faulty equipment is located. However, it is not always so simple. Often a faulty circuit may traverse one or more segments in which no alarms or other indications of failure are present. Another complicating factor is the possibility of multiple faults, some of which may mask others. Diagnosis requires careful reasoning from the known set of alarms and measured data, an understanding of the functional differences in each of the various equipment types, and knowledge about the network interconnection structure.

1.4 Summary

The next section describes the nature of system control problems which currently exist or may be anticipated in the future. Section three is a synopsis of the current programs in the evolutionary approach being pursued by the Rome Air Development Center. These programs are seeking engineering applications of machine intelligence to system control problems. In section four we provide a rationale for the use of machine intelligence as opposed to other, more traditional methods. Concluding observations are made in section five.

2. Problem Description

This section describes the system control problem in two distinct time frames: (1) the current and near-term context, and (2) a long-term context. In the near-term there are two major changes taking place which are having a significant impact on the ability of the various military departments to provide the highest quality tech control. These are a rapidly changing telecommunications technology and an erosion of the supply of senior manpower. The impact of these changes is described in section 2.1. For the long-term, system control will be most influenced by the implementation of these new technologies currently planned for the near-term and by the changing structure of the control system itself. These issues are addressed in section 2.2.

2.1 Near-term Problems

At present, system control is almost entirely a manual operation. Tech controllers have some automated equipment monitoring and automated test capability, but they do not have any automated support for fault isolation and diagnosis functions. A typical TCF may have 1000 or more circuits which either traverse or terminate at the local station. In addition, the tech controller may have responsibility for equipment at a remote, unmanned relay station. The current "data base" upon which the tech controller depends for detailed information about these circuits is a card file. Each card has the circuit data on one side, and often has a schematic diagram of the circuit on the reverse. Much of the expertise needed to identify and diagnose problems with circuits traversing older equipment resides only in the experience of the senior tech controllers.

The rapid change in telecommunications technology has resulted in modernization of station equipment at a much greater pace than can be matched by tech controller training

programs. The impact of this is that the "new" tech controller needs more on-the-job learning. Typically, this means the new controller learns about the operation and fault diagnosis procedures for sophisticated digital communications equipment only when a problem arises. Since the reliability of these equipments is usually quite good, the opportunities for on-the-job training are few and infrequent. Given the sophistication of some of these devices, it is simply not realistic to expect tech controllers to become expert in a short time period under these conditions.

The training problems are only exacerbated by an erosion of the senior tech controller manpower supply. The rapid growth of the commercial telecommunications industry has placed a premium on the skills possessed by military tech controllers. As a result these people are given significant economic incentives to leave the military for private industry at about the same time they are just beginning to become fully trained. Thus the pool of senior, "expert" tech controllers is dwindling.

2.2 The Long-term Environment

As the transition to an all digital system becomes complete, the nature of system control will also change. At the lower levels, individual equipments will incorporate more than simple built-in test capability and will include diagnostic capability as well, thereby allowing self-identification of failed units at the replacement unit level. The incorporation of automated digital patching systems (called DPAS, for Digital Patch and Access System) will allow an entirely different, and more responsive, approach to circuit restoration when outages do occur.

The changing control structure will also have an impact on system control problems and how they are solved. Perhaps the most significant of these changes is the introduction of the SRCF and the potential for correlation of monitored data and integration of control actions across transmission subsystems and networks. To take advantage of this situation, system controllers must have a broader scope of expertise. In this new environment, a system controller should be able to recognize a problem which arises in one part of the system, and predict the impact on the global system of various network or transmission control actions taken to alleviate this problem.

The combination of improved on-line monitoring, remote reporting of alarms to a central point within a subregion, and the need for correlation of data from both transmission subsystems and networks presents a problem of potential information overload for the human controller at the SRCF. During a major outage the volume of data being sent to the SRCF which the controller must examine, interpret, and act upon is too much for human decision making. Along with this, the controller has many alternative corrective actions which must be considered within this dynamic context. These factors make an accurate and timely assessment difficult for the human controller. It seems most unlikely, given the current problems with senior level manpower retention, that the functional requirements for system control could be adequately met by human controllers alone.

Another major challenge for system control in this longer term environment is to achieve an appropriate balance between distributed and centralized control. The SRCF has been introduced in recognition that survivability of network management can only be achieved by distributing the potential for network monitoring and control to several stations instead of concentrating it at a single, vulnerable site. However, distribution introduces new problems concerned with maintaining global coherence among these distributed controllers. An important question to be addressed is how might distributed control facilities (such as the SRCFs) work in a cooperative manner for the benefit of the system as a whole?

3. Evolutionary Approach

3.1 Background

The problem of designing and implementing an effective control system for a complex, evolving communications network is not new. The U.S. Air Force has had prior experience in at least two related projects. In both cases, an attempt was made to design a single, complete, fully automated control system as one major engineering effort. In both cases the products fell short of expectations. A major factor in both cases was the need from the control system designer's perspective to completely specify everything in advance while the target system was still evolving and thus could not be fully specified. The modular, evolutionary approach currently being followed is intended to avoid these same problems again.

3.2 Modular Development

The key aspect of this evolutionary approach is the development of incremental improvements in system control capability through a series of modular hardware/software systems. For example, previous work has already developed an automated transmission monitoring and control system (TRAMCON). While this represents only a "bare bones" degree of automation, it provides an essential service which can be used by more advanced systems which will come later. Similarly, as automated test systems are currently being developed, these too may be used by more advanced systems of the future as low level "extension tools" to test circuit segments suspected of degradation or failure. One of the benefits of this modular development is the usefulness of each of these additional systems. Since each new module is designed to provide an important functional capability independent of subsequent developments, failures or problems with more advanced systems do not disable the lower level modules.

A modular development approach offers other advantages as well. For example, it allows flexibility in adapting to changing system requirements. As each new modular component is developed, tested, and placed in the field, valuable feedback may be gathered from both the developers and the users which enhances the performance of subsequent components.

Another, pragmatic reason for a modular development process is budgetary. Traditionally communications systems rank behind weapon systems in seeking the attention of military and civilian budget planners. Funding for large, "do it all at once" programs is simply not available under present budget constraints. In particular, research involving the application of machine intelligence techniques, which is admittedly a high risk development area, must proceed by a carefully controlled sequence of demonstrably successful projects.

3.3 The Expert Tech Controller

The first system to be developed in this modular approach is an expert system to provide assistance to a tech controller in performing fault diagnosis. This system is intended to address the near-term system control environment. The initial version has been designed and implemented by Lincoln Laboratories and is described in detail in another paper in this session [2]. The objective of this effort is to capture the expertise of the most knowledgeable, senior tech controllers, and then build a knowledge-based system which could guide a less-skilled controller through a fault diagnosis procedure with the same proficiency as the senior "expert". The system was developed with the full cooperation of the TCF staff at Andrews Air Force Base, MD, and has been installed experimentally there since December, 1986. Initial reaction by TCF field personnel has been favorable.

Using the expert system building tool, ART, from Inference Corporation, Lincoln Lab researchers were able to quickly construct prototype versions which were then demonstrated to the expert tech controllers. These tech controllers provided valuable feedback on both the accuracy and useability of each successive version. The result is a system which incorporates both rule-based and procedural programming paradigms and emphasizes a natural human interface based on graphically displayed circuit diagrams.

3.4 Knowledge-based Control For The Defense Switched Network

The second project addresses system control needs at the boundary of near-term and long-term environments. The Defense Switched Network (DSN) is a circuit switched voice network designed to replace the older technology AUTOVON switches with new digital switches. The goal of this effort is to apply machine intelligence to the network control function for DSN. One specific objective is to demonstrate the use of a network simulator in the knowledge acquisition process and in validating the performance of a knowledge-based system as it is being developed. This is an important problem when intelligent control systems must be designed before the system to be controlled has been in operation -- and thus before a base of prior experience is available. It also represents a novel demonstration/testing approach for any system which can not be taken out of service or offline for testing purposes.

This project is currently in the initial stages of study by researchers at Lincoln Laboratories and work has concentrated on the network simulator. As the project proceeds, a dual processor architecture is envisioned. The simulator runs in one

processor in a UNIX environment, while the second processor supports the development of the knowledge-based DSN controller in a LISP world. Initially the interface between these two systems is a knowledge engineer/programmer. Using the simulation to develop an understanding of DSN operation, the engineer will develop rules and procedures which constitute the base of knowledge to be used by the DSN controller. For testing purposes the two systems will be directly connected, and the decisions made by the controller will act upon the simulated network. In this way the performance of the DSN controller may be observed under a variety of network operating scenarios. This configuration also provides a basis for further experimentation in the automated development of expert systems.

3.5 Cooperative Problem Solving For Distributed Network Management

The third research effort is an investigation of distributed problem solving. This effort, being performed by Clarkson University under a contract with the Northeast Artificial Intelligence Consortium (NAIC), is directed toward the long-term system control environment. Work in this area offers the hope of providing a means for building survivable networks of cooperating, intelligent agents which are able to interact in performing each of the various tasks required in system control. While the work builds on the results of the previous efforts, it is aimed at finding new problem solving paradigms suitable for a distributed set of agents. Each of these agents should be able to function independently of the others, yet be able to cooperate with any surviving subset to provide control of the maximum remaining network resources.

A diversely distributed architecture has been designed which supports both the geographic distribution of control facilities among SRCFs and the functional distribution of system control tasks [3-5]. This architecture is being implemented in a distributed AI system (DAISY) testbed. The DAISY testbed supports simulation of a distributed problem solving system on a group of heterogeneous LISP machines, in which the number of processors simulated exceeds the number of actual processors [6]. DAISY also includes a single user interface to a global knowledge base which employs a natural, graphics-based language. Partitioning of the knowledge base along subregional boundaries is performed by the system prior to beginning a simulation session, thus giving each simulated control facility a localized, private knowledge base. The specific problem solving agents for each control facility are currently under development.

4. Rationale

In this section we address the question, "Why AI?". From a researcher's perspective the obvious answer is, "Because it is an interesting problem domain." Distributed network management and control as a problem area has a number of characteristics which make it an appropriate domain for developing and testing new ideas in distributed problem solving. First, it is a real problem, not a toy, and has sufficient complexity to make it challenging. The problems which arise are not

intractable, yet they often require a substantial amount of time when solved by humans. There are no known algorithmic solutions, and exhaustive searching is not practical. Further, in the real world of dynamic operating conditions, with noisy and/or incomplete data, it is impossible to prepare in advance for all of the possible situations which may arise. Conventional approaches to automated system design do not permit operation under these conditions.

From the user's perspective, the development of machine intelligence applications offers the promise of providing a reliable, high-level assistant to system controllers. In most situations, a tech controller or network manager does not need more data. What is needed is help in interpreting the data, in recognizing critical situations quickly, and in sorting through the many possible corrective actions to find the best choice in the current context. These automated systems must work with the human decision maker in a natural manner. We are just beginning to be able to build systems which approach these goals, and the potential which AI techniques have to offer in developing these systems cannot be ignored.

5. Concluding Remarks and Acknowledgements

In this paper we have presented a high level view of an evolutionary approach to the development of machine intelligence applications for communications system control. This approach involves the development of both hardware and software modular components, each designed to provide an incremental improvement in overall system control capability. These systems also serve as a series of increasingly complex systems which together will make a significant advancement in the application of machine intelligence technology. While this program is still in the early stages of development, a prototype expert system has been implemented and placed in the field for experimental testing.

Finally, we also observe that although all the work described here has been specifically directed toward network management and control for the Defense Communications System, it is applicable to a much more broader range of problems. For example, the DSN controller work will involve fundamental research on issues of a much wider interest, such as the testing and evaluation of expert systems, and automated knowledge acquisition. The distributed AI research has wide ranging applications, not only to survivable communications networking in general, but also to other problems involving distributed resource allocation with decentralized knowledge and control.

The work on cooperative problem solving was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

References

1. "Defense Communications Operations Support System Subregional Control Facility Functional Requirements", Defense Communications Agency, April 1986. (Draft).
2. B. W. Otis and H. M. Heggestad, "The Expert Tech Controller: A Network Management Expert System", to be presented at MILCOM 87, Washington, D.C., October 1987.
3. S.E. Conry, R.A. Meyer, and J.E. Searleman, "A Shared Knowledge Base For Independent Problem Solving Agents," Expert Systems in Government Symposium, McLean, VA, October 24-25, 1985, pp. 178-186.
4. S.E. Conry, "Distributed Artificial Intelligence in Communications Systems", Expert Systems in Government Symposium, McLean, VA, October 22-24, 1986, pp. 286-289.
5. R.A. Meyer and C. Meyer, "The Role of Knowledge-based Systems in Communications System Control," SPIE Applications of Artificial Intelligence V Conference, Orlando, FL, May 18-20, 1987.
6. D.J. MacIntosh and S.E. Conry, "SIMULACT: A Generic Tool For Simulating Distributed Systems," Eastern Simulation Conference, Orlando, FL, April 6-8, 1987.



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.